Software Qual J (2018) 26:809–847
DOI 10.1007/s11219-016-9345-3

Integrating software quality models into risk-based testing

Harald Foidl[1] · **Michael Felderer**[1]

Published online: 12 November 2016

Abstract Risk-based testing is a frequently used testing approach which utilizes identified risks of a software system to provide decision support in all phases of the testing process. Risk assessment, which is a core activity of every risk-based testing process, is often done in an ad hoc manual way. Software quality assessments, based on quality models, already describe the product-related risks of a whole software product and provide objective and automation-supported assessments. But so far, quality models have not been applied for risk assessment and risk-based testing in a systematic way. This article tries to fill this gap and investigates how the information and data of a quality assessment based on the open quality model QuaMoCo can be integrated into risk-based testing. We first present two generic approaches showing how quality assessments based on quality models can be integrated into risk-based testing and then provide the concrete integration on the basis of the open quality model QuaMoCo. Based on five open source products, a case study is performed. Results of the case study show that a risk-based testing strategy outperforms a lines of code-based testing strategy with regard to the number of defects detected. Moreover, a significant positive relationship between the risk coefficient and the associated number of defects was found.

Keywords Risk-based testing · Software quality models · Software testing · Software quality · Software risk management · Test management · Test process improvement · Software process improvement · Case study

1 Introduction

Testing is an essential quality assurance technique for modern software-intensive systems which often has to be performed under severe pressure (Felderer et al. 2014b; Perry and Rice

✉ Michael Felderer
michael.felderer@uibk.ac.at

Harald Foidl
harald.foidl@uibk.ac.at

[1] University of Innsbruck, Innsbruck, Austria

1997). A challenging time schedule, limited resources (Felderer et al. 2014b), and increasing pressure from senior management, who often see testing as "something that has to be done" (Perry and Rice 1997), are the major causing factors for that. In addition, complete software testing is virtually impossible (Redmill 2004; Pries and Quigley 2010). As a result, effective and efficient software testing must be selective (Redmill 2004) to ensure the right amount of testing (Graham et al. 2008). Risk-based testing (Felderer and Schieferdecker 2014), which utilizes identified product risks of a software system for testing purposes, has a high potential to support and improve testing in this context (complete testing not possible, challenging time schedule and limited resources). It optimizes the allocation of resources and time, is a means for mitigating risks, helps to early identify critical areas, and provides decision support for deciding when to release (Felderer et al. 2014b).

Lately, the international standard ISO/IEC/IEEE 29119 Software Testing (2013) on testing techniques, processes, and documentation even explicitly mentions risks as integral part of the testing process. Also, several risk-based testing approaches which consider risks of the software product as the guiding factor to support decisions in all phases of the test process have been proposed in the literature (Erdogan et al. 2014; Felderer and Schieferdecker 2014).

A core activity in every risk-based testing process is risk assessment because it determines the significance of the underlying risk values and, therefore, the quality (effectiveness and efficiency) of the overall risk-based testing process (Felderer et al. 2012). Risk assessment is often done in an ad hoc manual way which is expensive, time-consuming, and has low reliability.

Because product risk can be seen as a factor that could result in future negative consequences (ISTQB 2015), which are usually system and software defects in the field of software testing (Redmill 2004), one can argue that risk represents missing software product quality and therefore should be measured via software quality assessment. The recent standard ISO/IEC 25010 (2011) decomposes software quality into characteristics which further consist of subcharacteristics and even sub-subcharacteristics. Quality Modeling and Control (QuaMoCo) operationalizes ISO/IEC 25010 by providing a tool-supported quality assessment method for defining and assessing software quality (Wagner et al. 2015; Deissenböck et al. 2011). Based on this quality model assessment, it is also possible to provide an objective and automation-supported risk assessment for risk-based testing.

As risk represents missing software quality and quality assessments based on quality models that already describe the quality-related risks of a whole software product (Zhang et al. 2006; Wagner 2013; ISO/IEC 25010 2011), this article addresses the research objective: *how quality assessments based on quality models can be used and applied in risk assessment for testing purposes.* This objective is investigated by showing how quality models can principally be used for risk assessment and by providing a concrete integration of a quality assessment based on QuaMoCo into risk-based testing and its evaluation.

An exploratory study of available risk-based testing approaches by Felderer et al. (2015) showed that, until now, the potential of quality models for risk assessment in the context of risk-based testing has not been investigated. The contribution of this article is, on the one hand, to show the potential usage of quality models for risk-based testing by presenting two integration approaches and, on the other hand, to provide a concrete integration including tool support and an empirical evaluation for the quality model QuaMoCo. In addition, the presented integration approach bridges the gap between the international standard ISO/IEC 25010 on Software Quality, which is operationalized by the quality model QuaMoCo, and the international standard ISO/IEC/IEEE 29119 on Software Testing which explicitly mentions risks as an integral part of the testing process.

The evaluation of the developed integration approach based on a case study of five open source products showed that a risk-based testing strategy outperforms a line of code-based testing strategy according to the number of classes which must be tested in order to find all defects. In addition, a significant positive relationship between the risk coefficient[1] and the associated number of defects of a class was found. Moreover, on average, 80 % of all defects of the five analyzed software products were found by testing 30 % of all classes when a risk-based testing strategy was applied. For the sake of comprehensibility and due to the fact that an explicit distinction is not always required, we use the term *defect* according to Wagner (2013) as a superset of faults (=bugs) and failures in this article. This is because there is always some relationship between the two, and at a certain abstraction layer, it is useful to have a common term for both.

The remainder of this article is structured as follows. Section 2 discusses background on risk-based testing and software quality models as well as related work on their integration. Section 3 presents two generic integration approaches of how quality assessments based on quality models can be used for further risk-based testing of the investigated software product. Section 4 presents the concrete integration of quality assessments and risk-based testing on the basis of the open quality model QuaMoCo. Section 5 describes the applied research design including the research questions, the case selection as well as the data collection, analysis, and validity procedures. In Section 6, the results of the case study and threats to validity are discussed. Finally, Section 7 draws conclusions and presents possible future work.

2 Background and related work

This section discusses background on risk-based testing and software quality models as well as related work about their integration. Section 2.1 provides background on risk-based testing and Section 2.2 on software quality models. Finally, Section 2.3 discusses related work on the integration of quality models and risk-based testing.

2.1 Risk-based testing

Risk-based testing (RBT) is a testing approach which considers risks of the software product as the guiding factor to support decisions in all phases of the test process (Gerrard and Thompson 2002; Felderer and Schieferdecker 2014). A *risk* is a factor that could result in future negative consequences and is usually expressed by its probability and impact (ISTQB 2015). In software testing, the *probability* is typically determined by the likelihood that a defect assigned to a risk occurs, and the *impact* is determined by the cost or severity of a defect if it occurs in operation. Mathematically, we can define the risk exposure (R) of an arbitrary risk item or asset (a) as a multiplication of the probability factor (P) and the impact factor (I):

$$R(a) = P(a) * I(a)$$

In the context of testing, a risk item is anything of value (i.e., an asset) under test, for instance, a requirement, a component, or a defect one explicitly wants to avoid. Risk exposure values are estimated during development before the information whether a risk item is actually defective or not is available. Based on the risk exposure values, the risk items are typically

[1] We assumed the impact factor of the risk coefficient to be constant in our case study.

prioritized and assigned to *risk levels*. The resulting risk information is used to support decisions in all phases of the test process.

For the determination of risk, probability and impact several proposals were made in the literature (Felderer et al. 2015). Probability of defect occurrence is often determined by technical factors, whereas impact is often determined by business factors. For instance, Van Veenendaal (2012) proposes complexity, number of changes, new technology and methods, size, or defect history as probability factors and critical areas, visible areas, most used areas, business importance, or cost or rework as impact criteria. In another paper, Felderer et al. (2012) propose, for instance, code complexity, functional complexity, or testability as probability criteria as well as importance or usage as impact criteria. All listed factors are typically estimated manually and not guided by software quality models (Felderer et al. 2015) as proposed in this article. However, the guidance of risk-based testing by software quality models is of high practical importance. In previous studies (Felderer and Ramler 2014a, b; Felderer and Ramler 2016), we found that making testing more effective in terms of (1) detecting additional defects in testing such that fewer defects slip through to the field as well as (2) prioritization for detecting most critical defects first to reduce the overall stabilization costs and time are essential benefits of risk-based testing. Both aspects of effectiveness can be addressed by suitable software quality factors (systematically) selected from quality models. Guidance by software quality models thus also supports the development of risk models for testing purposes in a structured way.

2.2 Software quality models

Software quality models are, according to Deissenböck et al. (2009), a well-accepted mean for managing and describing software quality as the study by Wagner et al. (2012a, b, c) showed. In the last 30 years, plenty of quality models were developed by various researchers to understand and measure the quality of software (Kitchenham and Pfleeger 1996; Deissenböck et al. 2009). A complete coverage of all research contributions, existing literature, approaches, and concepts in the area of software quality models would be out of the scope of this article. Therefore, the following subsection aims to provide a general understanding about software quality models and presents the ISO/IEC 25010 standard about software quality, which is operationalized by QuaMoCo, in more detail.

Based on the long history of research effort on quality models, they can be seperated into different groups, for example hierarchical and richer quality models (Wagner et al. 2015), meta-model-based and implict quality models (Wagner 2013) or basic and tailored quality models (Miguel et al. 2014). Further, Deissenböck et al. (2009) suggest to classify software quality models according to their different purposes.

The predominat group is the hierarchical quality models (Wagner et al. 2015) which decompose the concept of quality into different factors, criteria, and metrics (Factor Criteria Metrics models) (Cavano and McCall 1978). Examples are McCall's quality model (McCall et al. 1977), Boehm's quality model (Boehm et al. 1978), FURPS quality model (Grady 1992), QuaMoCo (Wagner et al. 2015), or the ISO/IEC 25010 quality model (ISO/IEC 25010 2011).

The ISO/IEC 25010 standard, as a hierarchical definition quality model, decomposes software quality into characteristics which further can consist of subcharacteristics and even sub-subcharacteristics (Wagner 2013). The aim of this decomposition is to reach a level where the characteristics can be measured in order to evaluate the software quality (Wagner 2013). In detail, the ISO/IEC 25010 (2011) defines two quality models, the quality-in-use, and the

product quality model to evaluate and define software quality (Wagner 2013). The product quality model uses eight characteristics for describing a software product's quality in a comprehensive way. Figure 1 graphically illustrates these eight charactertistics with their corresponding subcharacteristics.

Many different quality models were developed in the last decades (Wagner 2013). According to Al-Qutaish (2010), it is a real challenge to select which model to use. A comprehensive overview about the different quality models and concepts can be looked up in Al-Qutaish (2010), Miguel et al. (2014), or Wagner (2013, Chapter 2).

2.3 Approaches integrating risk-based testing and quality models

This section aims to review related work about quality models associated with risk-based testing. Research interest on software quality is according to Miguel et al. (2014) as old as software construction itself. Although the concepts of software quality models and risk-based testing were addressed by several research papers and contributions (i.e., Deissenböck et al. (2007), Franch and Carvallo (2003) or Felderer and Schieferdecker (2014)), we found no related work which explicitly deals with integrating quality models and risk-based testing.

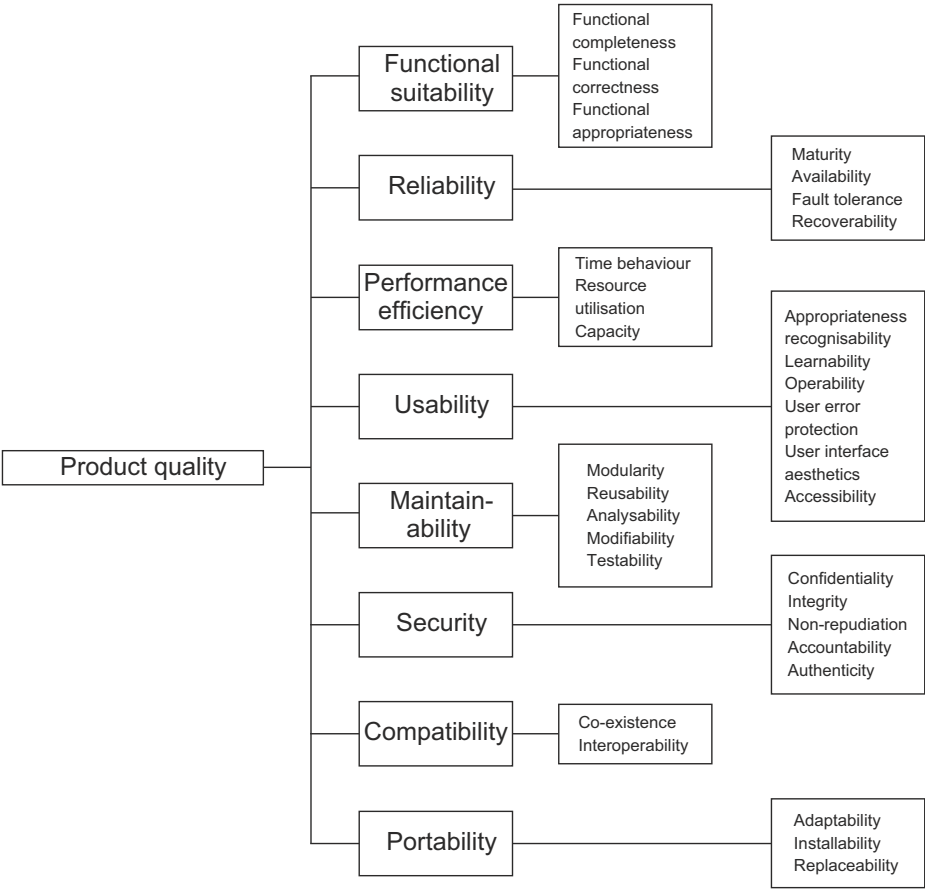


Fig. 1 Product quality model of ISO/IEC 25010 based on Wagner (2013)

Moreover, no existing risk-based testing approach in the literature especially considers quality-related information as basis for further execution of the risk-based testing process. Neither a tool was available in the literature that supports risk-based testing by using quality-related information. However, two contributions were identified in the literature which can be seen as related work as their contribution partially deals with integrating quality related information and software testing.

The first contribution (Neubauer et al. 2014) deals with the extension of the "Active Continuous Quality Control" (ACQC) approach (Windmüller et al. (2013)) for supporting risk-based testing. ACQC is an approach that aims to automatically maintain software test models by employing automata learning technology. Neubauer et al. (2014) extended the ACQC approach with risk analysts in order to support risk-based testing. Although the ACQC approach is not a quality model, we assumed this contribution as relevant as it actively integrates risk analysts in the ACQC approach in order to prioritize critical user interations with the software system.

The second contribution (Zeiss et al. 2007) adapted the ISO/IEC 9126-1 (2001) quality model to test specifications. Concretely, the authors developed a quality model for test specifications which is based on seven internal quality characteristics provided by the ISO/IEC 9126 domain. We assumed this contribution as important because the usage of quality models which instantiate test specifications seems to be promising for the integration with risk-based testing.

3 Integration of quality models into risk-based testing

In this section, we present two generic approaches how quality models can be integrated into risk-based testing. The presented approaches are based on previous work of Felderer et al. (2012) who defined a model-based risk assessment procedure integrated in a generic risk-based testing process.

The risk assessment procedure defined by Felderer et al. applies automatic risk assessment by static analysis. Therefore, Felderer et al. suggest the usage of automatic metrics for determining the probability and impact factor. Due the fact that metrics for the impact factor are typically derived from requirements and depend on the evaluator's viewpoint, they are usually evaluated manually. At least web applications could use Google Analytics (2005) to determine, for example, the frequency of use and therefore the importance of single parts of the system. Changing the look and feel of an online banking system's graphical user interface is typically not as often used as the function to transfer money. One further possibility to determine the frequency of use is to use earlier deployed versions of a software system. In the case that a software system is developed completely new and no previous versions are available, the frequency of use can be determined by analyzing similar software systems from the same category (i.e., web browsers, accounting software systems).

According to Van Veenendaal (2009, p. 9), determining the probability factor means predicting where the most defects are located in the software. The most defects are typically located in the worst areas of the software. Redmill (2004, p. 8) further suggests to observe the quality of the documentation and structure of the software code for determining the likelihood of defects. Moreover, Van Veenendaal (2009, p. 9) claims that one of the most important defect generators is complexity. For determining the complexity, a lot of different metrics (i.e., McCabe (1976)) are available (Van Veenendaal 2009, p. 9). Nagappan et al. (2006) found

that code complexity metrics are an effective mean to predict defects in software code. Further research showed (i.e., Catal et al. (2007), Jiang et al. (2008), Radjenovic et al. (2013)) that software metrics in general are useful for predicting defects and their location in software products. In the following, Felderer et al. (2012, p. 163) claim that the employment of different metrics, which can usually be determined automatically, can serve as a basis for determining the probability factor.

Current quality models use an integrated chain from rather abstract software characteristics down to specific metrics. Therefore, quality models typically decompose the concept of quality into different factors, criteria, and metrics (Factor Criteria Metrics models (Cavano and McCall 1978)). A quality assessment further evaluates and specifies the quality of a software product based on a defined Factor Criteria Metric hierarchy.

Based on the previous explanation and discussion in this section, one can argue that the probability factor can be used to integrate quality assessments based on quality models into risk-based testing. Due to the fact that the probability factor is computed mainly based on metrics (Felderer et al. 2012, p. 163) as well as considers complexity (Van Veenendaal 2009, p. 9) and the quality of the structure of the software product (Redmill 2004, p. 8), it seems appropriate to use the probability factor for integrating quality assessments, which are based on metrics that define the quality characteristics, into risk-based testing. In addition, also Redmill (2005, p. 13) mentions the possibility to use quality factors as "surrogates" for determining the probability factor.

Hence, our basic integration idea is to process the information from a quality assessment based on a quality model in a way that it represents the probability factor in the concept of risk-based testing. According to the explanation that the impact factor is mainly determined manually (Felderer et al. 2012, p. 166) and varies based on different possible perspectives (Redmill 2004, p. 7), it does not seem appropriate to integrate quality assessments based on quality models into risk-based testing by using the impact factor. Therefore, the impact factor must be determined manually in our two presented integration approaches.

Felderer et al. suggest assigning the probability factor to units or components of a software system. Units are the technical implementations of a software system and components can contain several units. As a result, the approaches presented in the following aim to integrate the quality assessments of quality models into risk-based testing by mapping the information provided by the assessments to an adequate probability factor for each component or unit. For the sake of comprehensibility, the following description of the two approaches uses the more generic term component.

3.1 Approach 1

First, a quality assessment based on the defined quality model is conducted for each component (the setting of the quality model (i.e., which quality factors are used) must be the same for each quality assessment).

The quality assessment of each component must then further be processed to an adequate probability factor for each component under test. Meaning the results of the quality assessments (which are represented by the quality factors at the highest level of the quality model hierarchy) are used to determine the probability factors for each component. Hence, components with high quality are assumed to have a low probability of defects and vice versa. The impact factor must further be determined for each component, and finally, the risk coefficient can be calculated for each component. Figure 2 illustrates the approach by an example. The

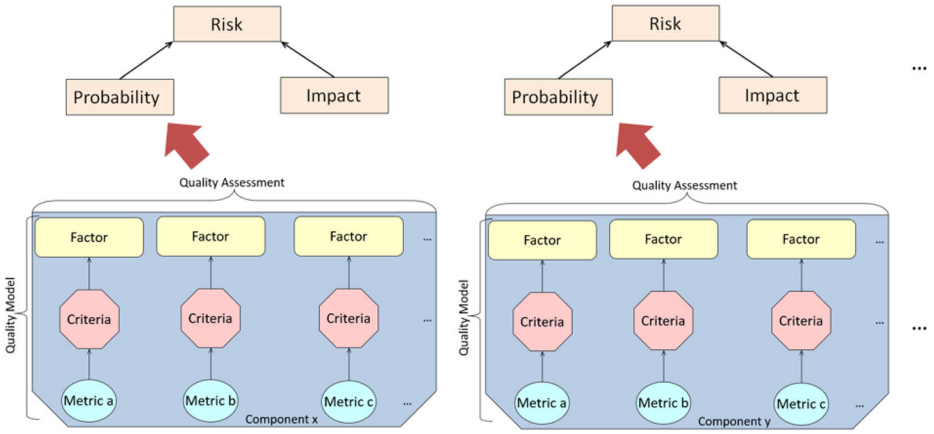


Fig. 2 Approach 1

rounded rectangles in Fig. 2 represent the quality factors, the octagons as the criteria, and the ellipsis as the metrics of the Factor Criteria Metric hierarchy. Further, the risk-based testing concept is represented by rectangles (Probability, Impact and Risk).

Components are illustrated by rectangles with cut corners. The red arrows represent the determination of the probability factor. In Fig. 2, a quality assessment is conducted for two components (component x and component y). Supposing the quality assessment specifies a high quality for component x and a low quality for component y, the probability factor for component x is then assumed to be lower as for component y because the high quality of component x indicates a low probability of defects. Further, the impact factor for both components must be determined. Finally, the risk coefficient for component x and y can be calculated by multiplying the probability and impact factor.

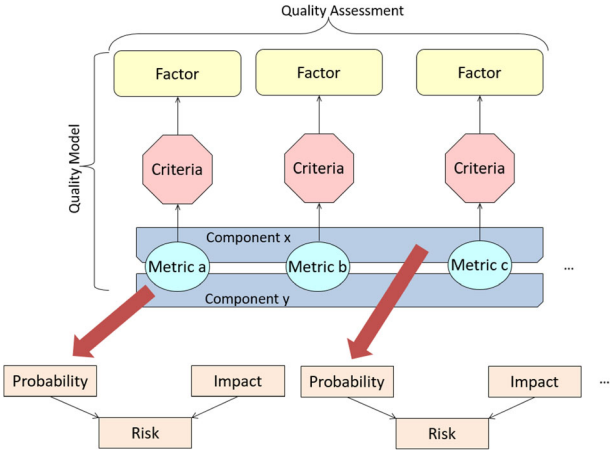
3.2 Approach 2

The second approach aims to directly use the metrics on the lowest level of the quality model hierarchy. Therefore, one single quality assessment for a software product is necessary and the measured values of each metric and component must further be processed to an adequate probability factor for each component.

Moreover, the impact factor must be determined for each component. Finally, the risk coefficient can be calculated by multiplying both factors. Figure 3 illustrates the approach by an example (the used symbols have the same meaning as defined above). Supposing there are three metrics (Metric a, Metric b, and Metric c) defined in the quality model and two components (component x and component y). Metric a, which measures the lines of code of each component, measures 340 lines of code for component x and 780 lines of code for component y. Metric b, which measures the nesting depth of each component, measures the values 16 for component x and 20 for component y. For the sake of comprehensibility, Metric c is skipped for the further illustration of the example.

Based on the measured values of each metric, an adequate probability factor can be calculated for each component. As a result, component y is assumed to have a higher probability factor as component x because the measured values indicate a higher complexity and therefore a higher probability of defects for component x.

Fig. 3 Approach 2



To summarize, the first difference between the two approaches is that approach 1 requires a *separate* quality assessment for each component of a software product and approach 2 only requires *one single* quality assessment for a software product no matter how many components the software product has. The second difference is that approach 1 directly uses the quality assessment results which are represented by the quality factors at the highest level of the quality model hierarchy to determine the probability factor. Approach 2 directly uses the metrics on the lowest level of the quality model hierarchy to determine the probability factor for each component. As the quality factors are based on the defined criteria and further on the assigned metrics, the defined aggregations and evaluations (i.e., which metric affects which criteria, how are the metrics aggregated) play a key role in approach 1 whereas approach 2 is not impacted by this (it uses only the metrics).

The decision, which of these two approaches should be applied, must be made for each concrete quality model. In the next section, we demonstrate the decision process and the integration with the open quality model QuaMoCo and compare both integration approaches.

4 Usage of QuaMoCo in risk-based testing

In this section, we present the integration of quality assessments and risk-based testing on the basis of the open quality model QuaMoCo. First, Section 4.1 introduces the main concepts of the open quality model QuaMoCo. Section 4.2 illustrates the application of the selected integration approach. Finally, Section 4.3 illustrates the implementation of the integration approach.

4.1 QuaMoCo

The open quality model QuaMoCo (Quality Modeling and Control) is an operationalized software quality model together with a tool chain containing a quality assessment method for defining as well as assessing software quality (Wagner et al. 2015; Deissenböck et al. 2011). The declared aim of QuaMoCo was to close the gap between generic and abstract software quality characteristics and concrete quality metrics (Wagner et al. 2012a, 2015). QuaMoCo

was developed by software quality experts from academia (Fraunhofer IESE[2] and Technische Universität München[3]) and industry (Siemens[4], SAP AG[5], Capgemini[6] and iestra[7]) (Wagner et al. 2015).

The main concept used in the QuaMoCo quality model is a factor. "A factor expresses a property of an entity" (Wagner et al. 2015, p. 104) whereas an entity "describe[s] the things that are important for quality" (Wagner et al. 2015, p. 104). The attributes of those entities (entities are for example a class, a method, an interface or the whole software product) are described by properties. For example, *Maintainability* would be a property of the entity *Software Product* and *Detail Complexity* a property of the entity *Class*. In order to bridge the gap between concrete metrics and abstract quality characteristics, Wagner et al. use the concept of a factor on two different levels of abstraction.

The first factor is named *Quality Aspect* and describes the abstract quality characteristics provided by the ISO/IEC 25010 (i.e., Maintainability, Security, Portability...). Quality aspects have the whole software product as their entity (i.e., Maintainability of the whole software product). *Product Factors* are the second type of factors and represent attributes (properties) of parts of the software product (i.e., Detail Complexity, Duplication). These two factors can both consist of several sub-aspects (in case of quality aspects) and sub-factors (in case of product factors). An important requirement regarding the leaf product factors is that they must be measurable. Therefore, Wagner et al. require them "to be concrete enough to be measured" (Wagner et al. 2015, p. 104). For example, the product factor *Detail Complexity of Method* can be measured by nesting depth and length. Further, the seperation of entities and their properties allows decomposing the product factors either regarding their entity or property. For example, the entity *Class* can be decomposed into the entities *Field* and *Method*.

This addresses the common problem of the difficult decomposition of quality attributes. In order to bridge the gap between the measurable properties of a software product and the abstract quality aspects, Wagner et al. set the abstract quality aspects in relation to the product factors. Concretely, product factors can either have a positive or negative *Impact* on quality aspects. For example, the presence of the product factor *Detail Complexity of Method* negatively affects the quality aspects *Analysability* and *Maintainability*. Further, the presence of the product factor *Conforming to Naming Convention of Class name* positively affects the quality aspects *Analyzability* and *Testability*.

For measuring the leaf product factors, Wagner et al. introduced the concept of a measure (metric). Although some authors (i.e., Pressman 2010, p. 614f) see a subtle difference between the terms metric and measure, we use the term metric in this article on behalf of both terms as suggested by Wagner (2013, p. 43). "A *measure [metric]* is a concrete description of how a specific product factor should be quantified for a specific context" (Wagner et al. 2015, p. 105). For example, the product factor *Detail Complexity of Method* is measured by the metric deep nesting. There can be multiple metrics for one product factor and a metric further can be used for quantifying multiple product factors. As "a concrete implementation of a measure [metric]" (Wagner et al. 2015, p. 105) *Instruments* are used. The metrics are seperated from their instruments in order to provide the possibility to collect data for metrics with different tools or manually.

[2] http://www.iese.fraunhofer.de/de/customers_industries/automotive/referenzprojekt_QuaMoCo.html
[3] https://portal.mytum.de/pressestelle/pressemitteilungen/news_article.2010-04-08.8067070550
[4] http://www.siemens.com/entry/cc/en/
[5] https://www.tu9.de/forschung/2183.php
[6] https://www.de.capgemini.com/news/QuaMoCo-projektabschluss
[7] http://www.itestra.de/expertise/research/forschungsprojekte/

For getting a complete quality evaluation of a software product, *Evaluations* are assigned to quality aspects and product factors. The evaluations consist of formula which aggregate the measured metrics from the instruments (for the product factors) as well as the evaluation results caused by the impacts of the product factors on the quality aspects. The left side of Fig. 4 illustrates the so far discussed quality model concepts. On the right side, some concrete quality aspects, product factors as well as metrics and instruments are shown. Here, it can be seen that the product factor *Detail Complexity* of the entity *Method* is measured by deep nesting which is further determined by the quality assessment tool ConQAT. In addition, the product factor *Duplication* of the entity *Source Code Part* is measured by clone coverage and clone overhead which are both determined by ConQAT. Both product factors negatively impact the quality aspects *Analysability* and *Modifiability* which are both sub-quality aspects of the quality aspect *Maintainability*. As follows, a more detailed example of a quality assessment is illustrated.

Example Figure 5 shows a limited quality assessment example of the Java Platform (version 6) with three metrics M_1*(#Doomed test for quality to NaN)*, M_2 (*#Lines of source code*), and M_3 (*Floating Point equality*) as well as one leaf product factor $F_{1.1}$ (*General expression applicability of comparison expression)*. The example is taken from Wagner et al. (2015, p. 110f).

The values of the three metrics are: $M_1 = 6$, $M_2 = 2\ 759\ 369$ and $M_3 = 9$. For ensuring the comparability across different software products, Wagner et al. defined normalization metrics (i.e., number of class, LoC) for normalizing the metrics. The normalization metrics were defined by two measurement experts for each metric (Wagner et al. 2012a, p. 1138). The metrics M_1 and M_3 in the example are normalized based on metric M_2 which results in the normalized metric $M_4 = \frac{M_1}{M_2} = 2.17E{-}6$ for M_1 and $M_5 = \frac{M_3}{M_2} = 3.19E{-}6$ for M_3. As a result of this normalization, the metrics M_1 and M_3 can be compared with other software products.

In the next step, the utility functions for M_4 and M_5 are defined whereas the utility of both is represented by a decreasing function. For specifying the utility, each metric has a linear

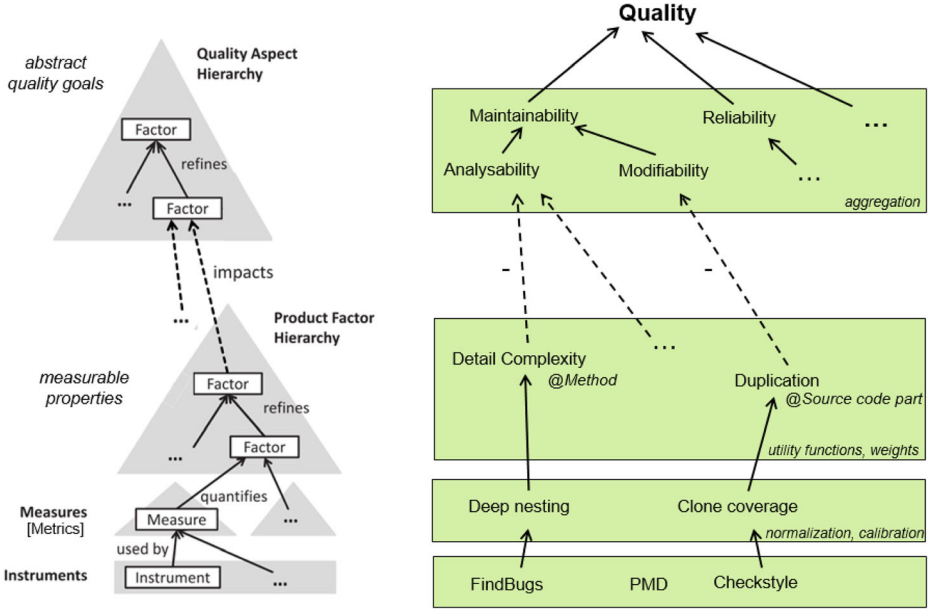


Fig. 4 Quality model concepts (adapted according to Wagner et al. (2015)

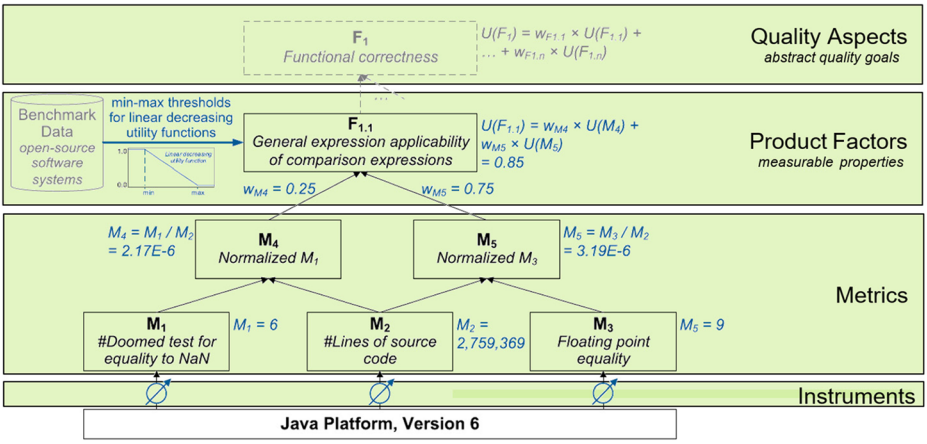


Fig. 5 Quality assessment example (adapted according Wagner et al. 2015)

decreasing or increasing utility function according to its associated leaf product factor (Wagner et al. 2012a, p. 1138). These utility functions provide a value between 0 and 1 whereas thresholds for the maximal utility (1) and the minimal utility (0) are determined by a benchmarking approach based on a large number of software products. The minimum and maximum thresholds for M_4 and M_5 are $\min(M_4) = 0$, $\max(M_4) = 8.50E-6$ and $\min(M_5) = 0$, $\max(M_5) = 3.26E-6$. Based on the defined thresholds, the utility values are 0.74 for M_4 ($U(M_4)$) and 0.89 for M_5 ($U(M_5)$). Figure 6 illustrates the utility function for metric M_4 with the two thresholds as well as the resulting utility value 0.74.

Finally, the utility values are aggregated based on their weights. The weights were assigned based on expert opinion or available data (Wagner et al. 2015, p. 111). The assigned weight for M_4 is 0.25 (w_{M_4}) and 0.75 (w_{M_5}) for M_5. This means the metric M_5 is three times more important for determining the leaf product factor *General expression applicability of comparison expression* ($F_{1.1}$) as metric M_4.

As a result, the aggregated utility value of this product factor ($U(F_{1.1})$) is $0.25 * 0.74 + 0.75 * 0.89 = 0.85$. For determining the higher level product factors as well as quality aspects (i.e., F_1 (*Functional Correctness*)), the same aggregation principle can be applied. As a last step, the aggregated utility values are mapped into a German ordinal school grade scale. The school grade scale provides a range from 1 (best grade) to 6 (worst grade), whereas the used thresholds are shown in Fig. 7.

For the application of the QuaMoCo approach, a tool chain which supports editing, building, and adapting quality models, assessing software products as well as visualizing the quality assessment results, was developed. The QuaMoCo tool chain[8] is freely available under the Apache license on the internet.

The tool chain consists of two main parts: the QuaMoCo quality editor and the quality assessment engine (Deissenböck et al. 2011). The aim of the quality editor is to provide the possibility to edit quality models by defining metrics, weights, or utility functions. The quality assessment engine automates the quality assessment procedure and is based on the toolkit ConQAT[9] (version 2013.10 used in this article). ConQAT is a quality assessment toolkit which

[8] http://www.QuaMoCo.de/
[9] https://www.cqse.eu/en/products/conqat/overview/

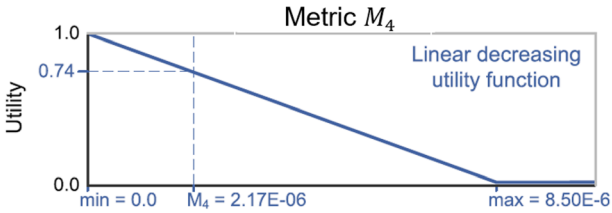


Fig. 6 Utility function (Wagner et al. 2015)

integrates several state-of-the-art code analysis tools (i.e., FindBugs, Gendarme, PMD, FxCop) and quality metrics.

4.2 Integration approach

This section presents the integration of the open quality model QuaMoCo and risk-based testing. We chose approach 2 for the integration of QuaMoCo and risk-based testing because the metrics in the QuaMoCo quality model were calibrated by benchmarking whole software products (Wagner et al. 2015) and are therefore not appropriate for the usage on single components (Approach 1).

Our integration approach limits to the programming language Java because we assumed to focus the integration approach on a specific programming language. Java was chosen because it ensures a huge repository of open source projects on which the integration approach can be applied. This means that the integration approach only uses quality assessments based on the Java quality model of QuaMoCo (Java module). An extension on other modules of QuaMoCo (i.e., C#) is planned for possible future work.

In the next step, the used components for which the risk coefficient should be calculated must be specified. We assumed classes of software products as components because, on the one hand, Java software products are typically structured hierarchically in packages and classes. On the other hand, QuaMoCo already provides the measured values for each metric on the class level.

As a result, the main principle of the integration approach is to analyze all metrics provided by the Java quality model of QuaMoCo based on their values according to each class in order to determine the probability factor of the risk-based testing concept. As already stated, the impact factor is not determined based on the quality assessment results of QuaMoCo and must be determined manually. We provide a suggestion for determining the impact factor manually. Afterwards, the determination of the probability factor is presented (Section 4.2.1). Further, a suggestion how to determine the impact factor in a manual way is outlined (4.2.2). Lastly, the final integration approach for determining the risk coefficient is presented (4.2.3).

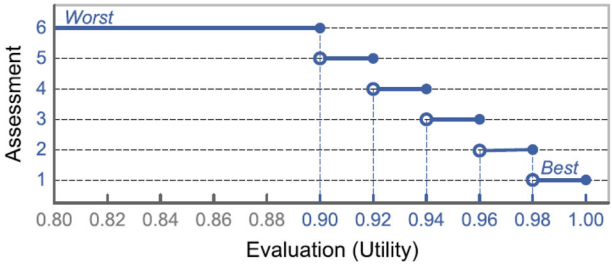


Fig. 7 Interpretation model (Wagner et al. 2015)

4.2.1 Determination of the probability factor

As a first step, we investigated all 23 metrics provided by QuaMoCo (Java module) and excluded those which were not appropriate for further usage (i.e., metrics which describe the overall software product and do not represent relevant information for a class, i.e., number of classes).

The remaining 13 metrics were divided in two groups. The first group (in the following referred as *Complexity Metrics*) is represented by 10 general metrics which measure properties of the source code directly for each class (i.e., nesting depth, number of methods, lines of code, etc.). On the other hand, the second group (in the following referred as *Rule Checking Metrics*) consists of metrics which are based on common rule checkers (i.e., FindBugs, Checkstyle, or PMD). These instruments aim to look for defects in the code, to find common programming flaws (i.e., empty catch blocks, unused variables, etc.), or to check if the code adheres to a coding standard (FindBugs 2003; PMD 2015; Checkstyle 2001). These instruments typically use defined rules for analyzing the code and provide their results as a list of findings. Concretely, QuaMoCo contains 361 rules/metrics for FindBugs, 4 rules/metrics for PMD, and 2 rules/metrics for Javadoc.

For further development of the integration approach, it is important to differentiate between these two groups. Table 1 presents the selected metrics for both groups.

The basic idea of determining the probability factor is to calculate one factor for the Complexity Metrics, which is named *Complexity* factor because all Complexity Metrics have in common that the higher their value is the more complex the associated class is.

Furthermore, one factor is calculated for each, the Javadoc findings (*Javadoc* factor), the PMD findings (*PMD* factor) as well as for the FindBugs findings (*FindBugs* factor). These four factors (Complexity, Javadoc, PMD, and FindBugs) are finally used to calculate the

Table 1 Quality assessment metrics

Metrics	Description
	Complexity metrics
#FieldDeclarations	Counts the number of field declarations for each class
#LocalVariableDeclarations	Counts the number of local variable declarations for each class
#Methods	Counts the number of methods for each class
#Statements	Counts the number of source statements for each class
#Types	Counts the number of types (i.e., classes, interfaces, enums) by counting the occurrences of the corresponding keywords for each class
Fan-In	Counts the number of classes depending on each class
LoC	Counts the lines of code
NestingDepth	Calculates the maximum nesting depth for each class
AvgMethodLenght	Calculates the average method length for each class
UnitCoverage	Probability that an arbitrarily chosen statement of a class is part of a clone
	Rule checking metrics
FindBugs Findings	Executes FindBugs rules and adds findings for rule violations
PMD Findings	Runs the specified PMD rules and adds findings for rule violations
Javadoc Findings	Analyses Javadoc comments in order to check if there is a Javadoc comment at all and uses a heuristic to check if a comment is sufficient. Findings are added for violations

probability factor. As follows, the calculation of the different factors is outlined. The calculation is based on the suggestion of Felderer et al. (2012) for calculating the risk coefficient. They propose to weight the used criteria for determining each factor.

Further, Felderer et al. (2012) suggest to use a range from 0 to 9 as a scale for the probability and also for the impact factor. The idea behind this range from 0 to 9 is that the probability can be seen as percentage, whereas the value 10 is skipped because we assume that no risk item fails for sure. Natural numbers between 0 and 1 are suggested to be used for the weights in order to see the weights as scaling factors.

Complexity metrics (*Complexity factor*) To calculate the complexity factor for each class, all measured values for each metric and class are analyzed. The lowest measured value of a metric is projected to 0 and the highest to 9. The remaining values are interpolated to values between 0 and 9. As a result, every complexity metric represents a value between 0 and 9 for each class. Afterwards, weights are added to each metric in order to represent its importance. These weights can be freely adapted according to the actual needs and the software product under investigation. Later, we provide concrete recommendations for the weights. In the next step, the interpolated values of all metrics for each class are summed up and divided by the sum of the weights in order to provide the *Complexity* factor for each class.

Example complexity factor In the following, an example illustrates this procedure. Table 2 shows the initial measured values for each metric and four classes. For example, class B has two methods and 43 lines of code. Class A has a nesting depth of 5 and 3 field declarations.

As a next step, the measured values for each metric are projected and interpolated on a scale between 0 and 9. The following formula illustrates this projection for the scale value $S(m, c)$ whereas m represents the metric and c the class:

$$\mathrm{S}(m,c) = (\text{measured value}(m,c) - \text{lowest value}(m)) * \frac{9}{(\text{highest value}(m) - \text{lowest value}(m))}$$

Table 2 Measured metric values

	Weight	Metric values			
		Class A	Class B	Class C	Class D
#FieldDeclarations		3	1	4	1
#LocalVariableDeclarations		17	1	21	1
#Methods		5	2	17	2
#Statements		78	6	42	3
#Types		1	1	5	1
Fan-In		1	0	2	0
NestingDepth		5	3	3	2
UnitCoverage		0	0	0	1
LoC		190	43	171	34

Table 4 Complexity factors

	Weight	Weighted interpolated metric values			
		Class A	Class B	Class C	Class D
#FieldDeclarations	0.5	3.00	0.00	4.50	0.00
#LocalVariableDeclarations	0.5	3.60	0.00	4.50	0.00
#Methods	0.8	1.44	0.00	7.20	0.00
#Statements	0.8	7.20	0.29	3.74	0.00
#Types	0.5	0.00	0.00	4.50	0.00
Fan-In	0.5	2.25	0.00	4.50	0.00
NestingDepth	1	9.00	3.00	3.00	0.00
UnitCoverage	0.8	0.00	0.00	0.00	7.20
LoC	1	9.00	0.52	7.90	0.00
	6.4	5.55	0.59	6.23	1.13

As an example for metric "LoC" and class C:

$$S(LoC, C) = (\text{measured} \quad \text{value}(LoC, C) - \text{lowest} \quad \text{value}(LoC)) * \frac{9}{(\text{highest} \quad \text{value}(LoC) - \text{lowest value}(LoC))} = (171-34) * \frac{9}{(190-34)} = 7.9$$

In the case that *highest value*(*m*) = *lowest value*(*m*) and thus to avoid a division by zero, the value 4.5 is assigned for $S(m, c)$. Table 3 shows the final projected values for each metric and each class.

The next step consists of adding weights (w), to each metric (m), and multiplying the projected values ($S(m, c)$) with the weights. The determination of the weights is described later.

$$S_w(m, c) = w(m) * S(m, c)$$

Table 3 Interpolated metric values

	Weight	Interpolated metric values			
		Class A	Class B	Class C	Class D
#FieldDeclarations		6.00	0.00	9.00	0.00
#LocalVariableDeclarations		7.20	0.00	9.00	0.00
#Methods		1.80	0.00	9.00	0.00
#Statements		9.00	0.36	4.68	0.00
#Types		0.00	0.00	9.00	0.00
Fan-In		4.50	0.00	9.00	0.00
NestingDepth		9.00	3.00	3.00	0.00
UnitCoverage		0.00	0.00	0.00	9.00
LoC		9.00	0.52	7.90	0.00

As an example for metric "LoC" and class C:

$$S_w(\mathrm{LoC}, \mathrm{C}) = w(\mathrm{LoC}) * S(\mathrm{LoC}, \mathrm{C}) = 1 * 7.9 = 7.9$$

Finally, all weighted values for each class are summed up and divided by the sum of the weights to calculate the *Complexity* factor for each class $C(c)$.

$$C(c) = \frac{\sum_{m=\text{first metric}}^{\text{last metric}} S_w(m, c)}{\sum_{m=\text{first metric}}^{\text{last metric}} w(m)}$$

As an example for class A:

$$C(A) = \frac{\sum_{m=\#\text{FieldDeclarations}}^{\text{LoC}} 3 + 3.6 + 1.44 + 7.2 + 0 + 2.25 + 9 + 0 + 9}{\sum_{m=\#\text{FieldDeclarations}}^{\text{LoC}} 0.5 + 0.5 + 0.8 + 0.8 + 0.5 + 0.5 + 1 + 0.8 + 1} = 5.55$$

Table 4 presents the final calculated complexity factors for each class.

As a result, every class is represented by a calculated complexity factor between 0 and 9.

$$C(A) = 5.55 \qquad C(C) = 6.23$$
$$C(B) = 0.59 \qquad C(D) = 1.13$$

Rule checking metrics (FindBugs, PMD, Javadoc factors) The Rule Checking Metrics consists of three different metrics (FindBugs findings, PMD findings, Javadoc findings). These three metrics are very similar according to their provided results. Each of them provides a set of findings for each investigated class. Hence, the usage of them for calculating the probability factor is similar. The main difference between them is that the metrics Javadoc findings and the PMD findings are limited according to the used categories or rules for inspecting the source code. In detail, the PMD findings provided by QuaMoCo uses the PMD ruleset "Unused Code Rules" which consists of four rules. In addition, the Javadoc findings applied by QuaMoCo consist of only two rules. In contrast, the FindBugs findings are based on more than 400 different rules. Although there are differing numbers of rules for these three metrics, the calculation of their factors (PMD, Javadoc, and FindBugs factor) is identical. As a first step, the findings for each rule are counted for each class and represented as percentage according to the total number of findings for a rule. As a next step, the percentages are projected and interpolated to a scale between 0 and 9. Now, the interpolated findings are multiplied with weights and finally summed up to build the corresponding factor for each class.

Example rule checking metrics For simplicity, the calculation in the following is shown in a general way for the four PMD rules. Table 5 shows the number of PMD findings for four classes A, B, C, and D.

Table 5 PMD findings

	Weight	#Findings				Total
		Class A	Class B	Class C	Class D	
UnusedLocalVariable	1	12	4	0	6	22
UnusedFormalParameter	1	6	4	1	10	21
UnusedPrivateField	0.5	12	4	6	1	23
UnusedPrivateMethod	0	0	0	0	0	0

Next, Table 6 presents the findings as percentage for each class. For calculating the findings as percentage ($F(m, c)$), the following formula is used, whereas m stands for the used PMD rule and c for the class.

$$\mathrm{F}(m,c) = \frac{\text{findings}(m,c)}{\sum_{c=\text{Class A}}^{\text{Class D}} \text{findings}(m,c)}$$

As an example for the rule "UnusedLocalVariable" and Class B:

$$\mathrm{F}(\text{Missing Documentation}, \mathrm{B}) = \frac{\text{findings}(\text{UnusedLocalVariable}, \mathrm{B})}{\sum_{c=\text{Class A}}^{\text{Class D}} \text{findings}(\text{UnusedLocalVariable}, \mathrm{B})} =$$

$$= \frac{4}{12+4+0+6} = 0.18$$

As a next step, the percentages are projected and interpolated to a scale between 0 and 9. This procedure is based on the same formula which was used for projecting the values for the *Complexity* factor. Table 7 presents the interpolated values.

Now, the interpolated findings are multiplied with weights, like in the calculation of the *Complexity* factor, and are summed up to determine the *PMD* factor for each

Table 6 PMD findings in percent

	Weight	#Findings in percent (0.01 = 1 %)				Total
		Class A	Class B	Class C	Class D	
UnusedLocalVariable	1	0.55	0.18	0.00	0.27	1.00
UnusedFormalParameter	1	0.27	0.19	0.05	0.49	1.00
UnusedPrivateField	0.5	0.53	0.17	0.26	0.04	1.00
UnusedPrivateMethod	0	0.00	0.00	0.00	0.00	0.00

Table 7 PMD interpolated values

	Weight	Interpolated values			
		Class A	Class B	Class C	Class D
UnusedLocalVariable	1	9.00	2.95	0.00	4.42
UnusedFormalParameter	1	4.50	2.86	0.00	9.00
UnusedPrivateField	0.5	9.00	2.39	4.04	0.00
UnusedPrivateMethod	0	0.00	0.00	0.00	0.00

class as shown in Table 8. How the weights are determined is described later in this section.

Finally, every class has a calculated *PMD* factor *PMD(Class)* between 0 and 9:

$$\mathrm{PMD}(\mathrm{A}) = 7.20$$

$$\mathrm{PMD}(\mathrm{B}) = 2.80$$

$$\mathrm{PMD}(\mathrm{C}) = 0.81$$

$$\mathrm{PMD}(\mathrm{D}) = 5.37$$

As already mentioned, the same calculations are used for determining the *Javadoc* factor *JD*(*Class*) and the *FindBugs* factor *FB*(*Class*). As a result, every class has three determined factors of type PMD, FindBugs, and Javadoc between 0 and 9.

Determination of the weights As a last step, the weights for each metric must be determined. The weights for the metrics of the *Complexity* factor are determined based on common literature on software complexity (i.e., Singh et al. (2011), Zhang (2009), Zimmermann et al. (2008), Zimmermann et al. (2007), Gyimothy et al. (2005), Jureczko (2011), Huang and Liu

Table 8 PMD weighted interpolated values

	Weight	Weighted interpolated values			
		Class A	Class B	Class C	Class D
UnusedLocalVariable	1	9.00	2.95	0.00	4.42
UnusedFormalParameter	1	4.50	2.86	0.00	9.00
UnusedPrivateField	0.5	4.50	1.19	2.02	0.00
UnusedPrivateMethod	0	0.00	0.00	0.00	0.00
	2.5	7.20	2.80	0.81	5.37

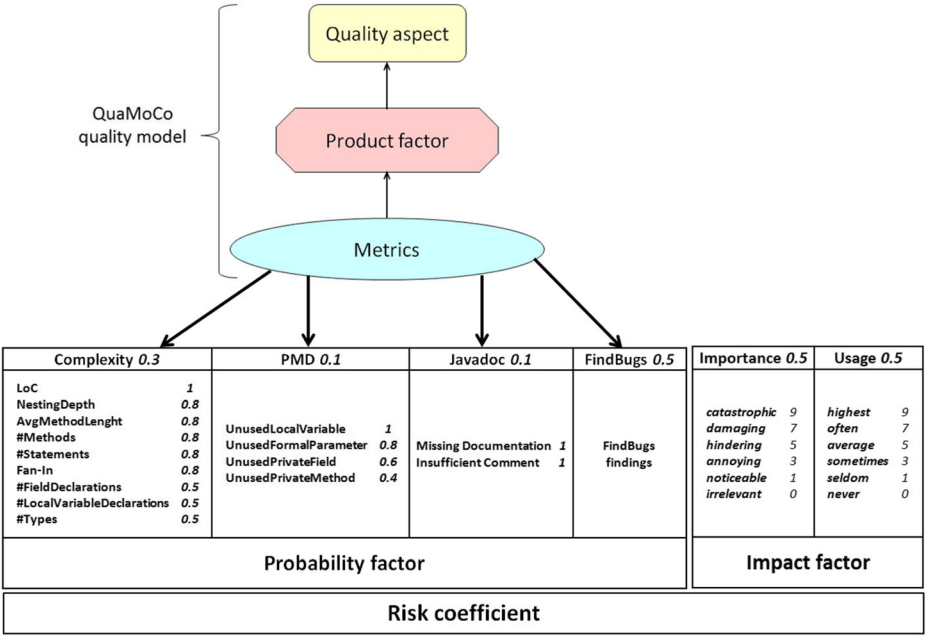


Fig. 8 Integration concept

(2013), Krusko (2003), Basili et al. (1995), Radjenovic et al. (2013)). Metrics suggested by literature which are highly relevant for predicting defects were weighted higher than the others.

Hence, metrics which are highly relevant for predicting defects were weighted with 1 (only the Lines of Code metric got an assigned weight of 1 because it was mentioned significantly more often in the literature as other metrics). Metrics which were mentioned by at least one author or contribution were weighted with 0.8. Lastly, metrics which were not mentioned were weighted with 0.5.

The idea of using these values was that metrics which were highly relevant for predicting defects are at least two times more important than metrics which were not mentioned by any author or contribution (1 and 0.5). The weight 0.8 was chosen because we think that metrics which were mentioned by at least one author or contribution rather should get a weight near to metrics which are highly relevant (1) than to metrics which were not mentioned by any author or contribution (0.5). Table 9 shows metrics which were mentioned by at least one author or

Table 9 Metrics relevant for predicting defects

Metric	Description
Lines of Code	Total lines of code
#Methods per Class	Number of methods per class
Nesting depth	Number control flow nesting levels (switch, else, do, etc.)
Coupling between Objects	Number of classes coupled (field access, method call, etc.) to a given class
#Statements	Count of semicolons in a file
#Line comments	Number of line comments

contribution. The assigned weights for each metric of the probability factor are shown in Fig. 8.

Due to the fact that the FindBugs instrument uses a huge base of rules, an analysis of ten open source Java projects (Table 10) was conducted in order to determine the most violated rules for FindBugs.

Concretely, these FindBugs rules are provided as a standard recommendation for determining the *FindBugs* factor. Therefore, these rules have an assigned weight of 1. Other FindBugs rules which are violated can be freely added to the set of FindBugs rules and must be weighted by the user of the further implemented tool support (standard weight 0.5).

The 12 most violated FindBugs rules which got an assigned weight of 1 are shown in the following Table 11. Further, all ten software products were analyzed according to the occurrence of the four PMD rules. Based on the results, the weights for the four PMD findings were assigned according to their frequency of occurrence. Due the fact that the Javadoc instruments only uses two rules (missing documentation and insufficient comment), no analysis was conducted. Based on the result of a study conducted by Dixon (2008), who states that both the number of comments and their quality (i.e. no @return tag in Javadoc) are effective predictors for defects, both are weighted with 1.

Final probability factor The final probability factor is calculated by summing up each of the four determined factors (*Complexity*, *Javadoc*, *PMD*, and *FindBugs* factor). Each factor has a weight in order to reflect its importance. For determining the weights, we used the frequency of their usage in the QuaMoCo quality model and recommendations of literature in the field of software complexity. Due to fact that the QuaMoCo quality model mainly is based on FindBugs rules, we recommend to assign a weight of 0.5 to the *FindBugs* factor.

Several contributions in the literature (i.e., Zhang (2009), Singh et al. (2011)) state that complexity and size are important predictors for software defects; thus, we recommend a weight for 0.3 for the *Complexity* factor. A weight of 0.1 is assigned to both the *Javadoc* and *PMD* factor because they only provide two (Javadoc) and four (PMD) rules for analysis of the source code and have therefore not the expressiveness as the other two factors. Of course, all weights are just recommendations of us and can be manually adjusted by test managers,

Table 10 Software products

Software product	Version / release	LoC	Size
OpenProj	1.4	148.264	Big
Sweet Home 3D	3.0	85.139	Medium
RSSOwl	1.2.4	82.258	Medium
Checkstyle	4.4	46.240	Medium
Log4j	1.2.16	43.018	Medium
Apache Tomcat	8.0.22	7.976	Small
TightVNC	1.3.10	6.874	Small
FCKeditor.java	2.6	5.187	Small
Twinkle	2.0	2.792	Small
OOMRM	1.5	663	Very small

Table 11 Most violated FindBugs rules

FindBugs Rule	Description	#Findings
DLS_DEAD_LOCAL_STORE	Dead store to local variable	92
DM_NUMBER_CTOR	Method invokes inefficient Number constructor; use static valueOf instead	68
MS_SHOULD_BE_FINAL	Field isn't final but should be	51
DM_DEFAULT_ENCODING	Reliance on default encoding	50
EI_EXPOSE_REP2	May expose internal representation by incorporating reference to mutable object	42
EI_EXPOSE_REP	May expose internal representation by returning reference to mutable object	41
SE_BAD_FIELD	Non-transient non-serializable instance field in serializable class	37
MS_PKGPROTECT	Field should be package protected	35
ST_WRITE_TO_STATIC_FROM_INSTANCE_METHOD	Write to static field from instance method	34
SIC_INNER_SHOULD_BE_STATIC	Should be a static inner class	33
URF_UNREAD_FIELD	Unread field	33
SF_SWITCH_NO_DEFAULT	Switch statement found where default case is missing	31

engineers, and other users based on their actual needs and characteristics of the investigated software products. Finally, the probability factor (P) for a class (c) can be computed as follows:

$$P(c) = 0.3*C(c) + 0.1*\mathrm{JD}(c) + 0.1*\mathrm{PMD}(c) + 0.5*\mathrm{FB}(c)$$

4.2.2 Determination of the impact factor

As stated earlier, the impact factor is not appropriate for being determined by the quality assessment results provided by QuaMoCo. According to the literature (Felderer et al. 2012, p. 166), the impact factor must be determined manually by product managers or the customers themselves based on the requirements documents (Felderer et al. 2012, p. 173).

For the sake of comprehensibility, the impact factor in our integration approach is described by two criteria suggested by Felderer et al. (2012), namely *Importance* and *Usage*. The usage describes the frequency of use by users (Felderer et al. 2012), whereas the importance criterion is defined according to Van Veenendaal (2009) as criterion which describes the most critical areas according to cost and consequences of defects. As suggested by Felderer et al. (2012), the impact factor is also determined by using a scale from 0 to 9. In addition, also the usage frequency (Usage criterion) and the defect consequence (Importance criterion) are rated with a scale from 0 to 9. The used scale for determining the *Usage* criterion is based on Felderer et al. (2012, p. 178). They provide a scale with textual values (used seldom, sometimes, average, often, highest) which are projected to numeric values. We added the textual value "never" in order to utilize the whole range of the scale. For determining the *Importance* criterion, a scale provided by Van Veenendaal (2009, p. 8) is used as a basis. The scale assigns values to possible consequences of defects (defect

is irrelevant, annoying, hindering, damaging or even catastrophic). Compared to the scale provided by Van Veenendaal, we added the consequence "irrelevant" to utilize the whole range of the scale. The scales (textual as well as their corresponding numeric values) used to rate the usage and importance criteria are illustrated in Fig. 8.

In case of huge software products, the rating should be done on the package or component level. The subordinate classes are then rated with the same value as their super ordinate package or component.

These two criteria are used as factors to determine the impact factor. The weight, representing the importance of each factor, must be determined manually by product managers or the customers themselves for each individual software product (Felderer et al. 2012). A weight of 0.5 for both factors is suggested by us as a standard value. Considering the two factors *Importance* (*Impo*) and *Usage* (*U*), the impact factor (*I*) can be calculated as following, where *c* stands for the corresponding class

$$I(c) = 0.5*\mathrm{Impo}(c) + 0.5*U(c)$$

4.2.3 Determination of the risk coefficient

After the probability and impact factor are determined, the final risk coefficient can be calculated by multiplying the probability and impact factor ($R(c) = P(c) * I(c)$).

As a result, every class has an assigned risk coefficient which indicates its risk. Now, the classes can be sorted according to the risk coefficient, as suggested by Felderer et al. (2014a). This ensures the possibility of starting the software testing procedure with those classes which correspond to the highest risk coefficients. Figure 8 illustrates the final integration approach including all used metrics and factors as well as weights for determining the probability and impact factor in order to computate the risk coefficient.

4.3 Tool implementation

This section presents the implementation of tool support for the integration approach described in the previous section. The tool was developed with Eclipse Luna[10] and the programming language Java[11]. The aim was to develop a platform-independent, extendible, and customizable tool support which can be used for computing the risk coefficient based on any quality assessment conducted with QuaMoCo (module Java).

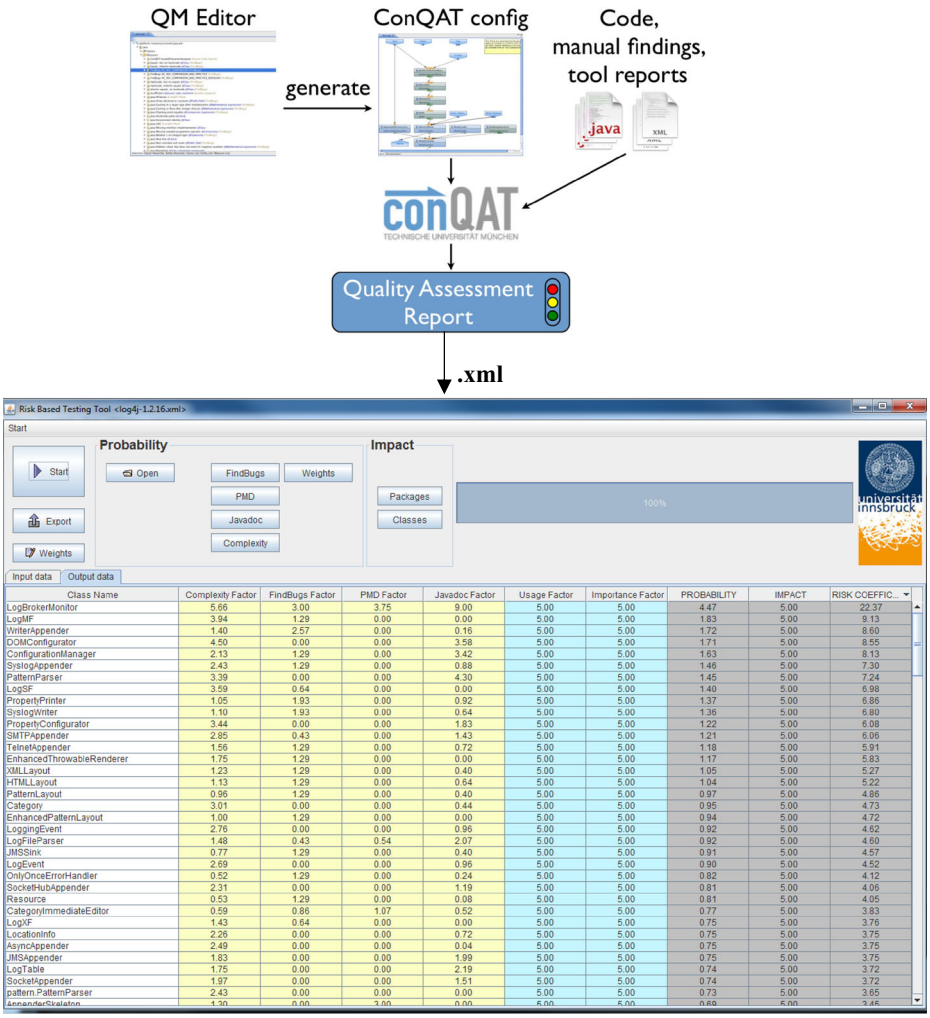
Figure 9 shows the QuaMoCo tool chain with its main components. On the left side, the QuaMoCo quality editor is shown which generates the ConQAT analysis configuration. The toolkit ConQAT then uses this configuration, the source code as well as optional findings of manual assessments to generate the final quality assessment report. QuaMoCo provides the possibility to export the quality assessment report as an XML file. The tool support uses this XML file to calculate the different factors. All different weights and factors can be changed and customized in the tool. After the computation is completed, the tool shows all classes of the investigated software product ranked by the final risk coefficient. Furthermore, the probability and impact values as well as the values of each calculated factor are shown in

[10] https://eclipse.org/luna/
[11] https://www.java.com/en/

separate columns. Note that the final risk coefficient is normally only relevant for ranking, but its difference to other risk coefficients cannot directly be interpreted. For this purpose, the decomposition into factors is provided. Moreover, the results can be exported as an Excel file for possible further usage. One can now start testing with those classes with the highest risk coefficient. Figure 9 illustrates this procedure and shows a screenshot of the results based on a quality assessment of the testing framework JUnit. The implemented tool is available online at http://bit.ly/1QGhGwJ.

5 Study design

To investigate the effectiveness of the developed integration approach, we conducted a case study on five different open source software products. This section presents the applied research



Class Name	Complexity Factor	FindBugs Factor	PMD Factor	Javadoc Factor	Usage Factor	Importance Factor	PROBABILITY	IMPACT	RISK COEFFIC...
LogBrokerMonitor	5.66	3.00	3.75	9.00	5.00	5.00	4.47	5.00	22.37
LogMF	3.94	1.29	0.00	0.00	5.00	5.00	1.83	5.00	9.13
WriterAppender	1.40	2.57	0.00	0.16	5.00	5.00	1.72	5.00	8.60
DOMConfigurator	4.50	0.00	0.00	3.58	5.00	5.00	1.71	5.00	8.55
ConfigurationManager	2.13	1.29	0.00	3.42	5.00	5.00	1.63	5.00	8.13
SyslogAppender	2.43	1.29	0.00	0.88	5.00	5.00	1.46	5.00	7.30
PatternParser	3.39	0.00	0.00	4.30	5.00	5.00	1.45	5.00	7.24
LogSF	3.59	0.64	0.00	0.00	5.00	5.00	1.40	5.00	6.98
PropertyPrinter	1.05	1.93	0.00	0.92	5.00	5.00	1.37	5.00	6.86
SyslogWriter	1.10	1.93	0.00	0.64	5.00	5.00	1.36	5.00	6.80
PropertyConfigurator	3.44	0.00	0.00	1.83	5.00	5.00	1.22	5.00	6.08
SMTPAppender	2.85	0.43	0.00	1.43	5.00	5.00	1.21	5.00	6.06
TelnetAppender	1.56	1.29	0.00	0.72	5.00	5.00	1.18	5.00	5.91
EnhancedThrowableRenderer	1.75	1.29	0.00	0.00	5.00	5.00	1.17	5.00	5.83
XMLLayout	1.23	1.29	0.00	0.40	5.00	5.00	1.05	5.00	5.27
HTMLLayout	1.13	1.29	0.00	0.64	5.00	5.00	1.04	5.00	5.22
PatternLayout	0.96	1.29	0.00	0.40	5.00	5.00	0.97	5.00	4.86
Category	3.01	0.00	0.00	0.44	5.00	5.00	0.95	5.00	4.73
EnhancedPatternLayout	1.00	1.29	0.00	0.00	5.00	5.00	0.94	5.00	4.72
LoggingEvent	2.76	0.00	0.00	0.96	5.00	5.00	0.92	5.00	4.62
LogFileParser	1.48	0.43	0.54	2.07	5.00	5.00	0.92	5.00	4.60
JMSSink	0.77	1.29	0.00	0.40	5.00	5.00	0.91	5.00	4.57
LogEvent	2.69	0.00	0.00	0.96	5.00	5.00	0.90	5.00	4.52
OnlyOnceErrorHandler	0.52	1.29	0.00	0.24	5.00	5.00	0.82	5.00	4.12
SocketHubAppender	2.31	0.00	0.00	1.19	5.00	5.00	0.81	5.00	4.06
Resource	0.53	1.29	0.00	0.08	5.00	5.00	0.81	5.00	4.05
CategoryImmediateEditor	0.59	0.86	1.07	0.52	5.00	5.00	0.77	5.00	3.83
LogXF	1.43	0.64	0.00	0.00	5.00	5.00	0.75	5.00	3.76
LocationInfo	2.26	0.00	0.00	0.72	5.00	5.00	0.75	5.00	3.75
AsyncAppender	2.49	0.00	0.00	0.04	5.00	5.00	0.75	5.00	3.75
JMSAppender	1.83	0.00	0.00	1.99	5.00	5.00	0.75	5.00	3.75
LogTable	1.75	0.00	0.00	2.19	5.00	5.00	0.74	5.00	3.72
SocketAppender	1.97	0.00	0.00	1.51	5.00	5.00	0.74	5.00	3.72
pattern.PatternParser	2.43	0.00	0.00	0.00	5.00	5.00	0.73	5.00	3.65
AppenderSkeleton	1.30	0.00	3.00	0.00	5.00	5.00	0.69	5.00	3.45

Fig. 9 Tool support

design which follows the guidelines for conducting and reporting case study research proposed by Runeson and Höst (2009). We first present the research questions (Section 5.1) addressed in this article. Afterwards, we illustrate the case selection (Section 5.2), data collection, analysis, and validity procedures to answer the research questions (Sections 5.3 to 5.5).

5.1 Research questions

Section 4 presented an integration approach of quality assessments and risk-based testing on the basis of the quality model QuaMoCo. The integration approach is limited to the programming language Java and focuses on the risk assessment of classes. As a result, all classes of the software product under test have an assigned risk coefficient which can be used for further test prioritization. In addition, the integration approach was implemented in a tool (see Section 4.4). Typically, a user of the tool support starts testing with those classes with the highest risk coefficients (risk-based testing strategy). Therefore, the developed integration approach works effectively if the classes with the highest risk coefficients of a software product are those classes where the most defects are located and the consequence of a defect in these classes is higher than in classes with lower risk coefficients.

In order to provide further empirical evidence for researchers and get relevant insights for practitioners, the developed integration approach is investigated on the basis of the following two research questions.

(RQ1) Is there a relationship between the risk coefficient and the number of defects of a class? This research question investigates the relationship between the ranking of all analyzed classes based on the risk coefficient and the ranking of all analyzed classes based on the number of defects.

(RQ2) How is the performance of a risk-based testing strategy compared with a line of code-based testing strategy? As research showed (i.e., Zimmermann et al. (2007, 2008), Gyimothy et al. (2005), Jureczko (2011)), lines of code act as a good predictor for defects in classes. Therefore, this research questions compares a test strategy based on the risk coefficient (starting testing with classes with the highest risk coefficients) with a test strategy based on the lines of code (starting testing with classes with the most lines of code) according to their performance (testing the "right" classes (those with defects) first)?

To answer these two research questions, the developed integration approach is applied for selected releases of five software products. First, a quality assessment based on QuaMoCo is conducted for these selected releases. Grounded on these quality assessments, the tool support is used to apply the integration approach in order to calculate risk coefficients for each class of the selected releases. These calculated risk coefficients of each class are then further analyzed according to the number of corresponding defects which were reported until the latest stable release in order to answer the research questions.

As we had no sufficient knowledge to determine the impact factor for the different classes and components of each software product, we assumed the impact factor to a constant value (5) for all classes in this study. Hence, the risk coefficient in this study is the result of the probability factor multiplied with a constant value. This means the risk coefficient in our study is just a multiple of the probability factor and therefore describes the probability whether a class is defect-prone or not.

5.2 Case selection

The following five open source software products were selected as units of analysis in our case study: JUnit[12], Mockito[13], Apache Commons IO[14], Apache PDFBox[15], and Google Guava[16]. The five software products were selected because they fulfill the requirements of providing source as well as binary files and all necessary libraries for compilation. Table 12 shows the software products with their latest stable release, the lines of code and a short description. All five software products and their releases are hosted on GitHub. GitHub[17] currently describes itself as the "world's largest code host" (GitHub Inc. 2008b) and is the single largest host for Git[18] repositories (Chacon and Straub 2014, p. 195). Git is a free and open source distributed version control system (Git 2005) which provides several different features for maintaining and sharing software code (Chacon and Straub 2014). A repository is a directory where Git stores code, text or image files of projects (Orsini 2013). The words "version" and "release" are used as synonyms in the further course of this article.

5.3 Data collection procedure

The case study is based on the source code and the information about defects of the selected software products. To determine the number of defects for each software product's class, features provided by GitHub were used. One feature is that GitHub uses a bug tracker called "Issues" where tasks and defects are tracked. A further feature provided by Git is the ability to "tag" specific points in history of a software code. Therefore, the code of different releases and versions can be tagged. This makes it comfortable to gather the code of different software versions and releases. In addition, every change in the software code is represented by a "commit." A commit provides, besides the information of the change, also additional information like the commit's author, the date of the commit, the commit message, and the changed files (classes). Therefore, we decided to use the commits provided by GitHub in order to determine the defects of each software product's class.

As a first step, we had to select a release in the past of each software product in which the developed integration approach could be applied and the risk coefficients could be calculated by the tool support. For determining this release in the past, we tried to ensure that the same number of major releases were between this selected release in the past and the latest stable release for all five software products. Due to the fact that each software product used another procedure for defining releases (i.e., 1.0, 1.1 or 1.1.1, 1.1.2), it was not explicitly clear for each software product to select the release in the past. To solve this problem, we chose the release in the past based on two prerequisites. First, the chosen release in the past had to provide source as well as binary files or had to be at least compileable to get the binaries. For conducting the necessary quality assessment with QuaMoCo, source and binary files of the software product are needed. Second, the chosen release in the past had to ensure a minimum of ten commits between this release and the latest stable release. Based on these prerequisites, the following

[12] http://junit.org/
[13] http://mockito.org/
[14] https://commons.apache.org/proper/commons-io/
[15] https://pdfbox.apache.org/
[16] https://code.google.com/p/guava-libraries/
[17] https://github.com/
[18] https://git-scm.com/

Table 12 Selected software products

Software product	Latest stable release	LoC	Description
Apache Commons IO	2.1	23.455	IO library for java
Apache PDFBox	1.8.9	135.658	Java library for pdf documents
Google Guava	18.0	128.302	Utility and collection library for java
JUnit	4.12	38.005	Testing framework
Mockito	1.10.19	23.297	Testing framework

releases in the past (Column "Selected Release" in Table 13) and the following range of releases to determine the defects (column "Releases Defects" in Table 13) were chosen for the case study.

We then determined the defects for each class by mining each commit in the defined range of releases, as Fig. 10 illustrates. Every commit was automatically checked whether the word "bug" was in the commit header or the commit message. The term "bug" was used because a large-scale study on tens of thousands of GitHub software projects (Bissyande et al. 2013) found out that the most popular tag used in GitHub issue reports is bug. Hence, we think that most programmers also use the term bug to commit defects. In the next step, the commits were manually analyzed and commits which did not indicate a defect (i.e., de*bug*ging related commits or commits which address an issue which is not related to a defect) were removed. From the remaining commits which indicate a defect, the number of defects for each class was determined by counting how often each class was listed in the selected range of releases. Finally, a list of all classes of the selected releases (for each software product) with its associated number of defects was available.

Classes which were added to the software product after the selected release were excluded from the analysis. For example, release 10.0 is selected for conducting a quality assessment and further for analysis by the tool, containing classes A, B, C, and D. Class E is added in release 11.0 and is associated with a defect. In case, the latest stable release is 18.0, and class E is listed as a defect-prone class. Since class E did not exist in the selected and analyzed release 10.0, there is no computed risk coefficient for this class. Therefore, classes which did not exist in the selected release were excluded from the analysis.

As a next step, a quality assessment of the selected releases in the past of each software product had to be conducted by QuaMoCo to provide the necessary XML file which is needed by the tool support to calculate the corresponding risk coefficients of each class. The XML file provided by the quality assessment was then imported in the tool support in order to calculate the risk coefficients for each class of the selected releases. Finally, all classes of the selected

Table 13 Software versions

Software product	Selected release	Releases DEFECTS
Apache Commons IO	1.4	1.4–2.1
Apache PDFBox	1.0.0	1.0.0–1.8.9
Google Guava	10.0	10.0–18.0
JUnit	4.6	4.6–4.12
Mockito	1.0	1.0–1.10.19

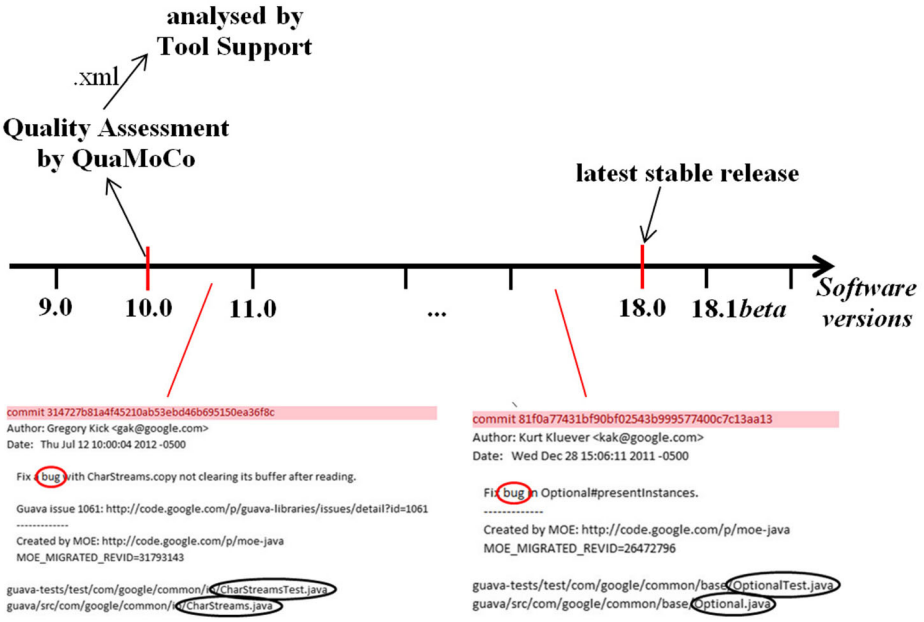


Fig. 10 Procedure

releases had a calculated risk coefficient which can be used for further analysis with the corresponding number of defects for each class.

Figure 10 graphically sketches this procedure exemplarily for the software product Google Guava. Here, the range of selected releases starts by release 10.0 (selected release) and ends with release 18.0. All commits which were committed in this range of releases are analyzed if the word "bug" is in the commit header or message. The resulting commits are further analyzed manually if they are associated with real defects. Commits which did not indicate a defect are excluded for the further analysis. In the next step, the number of defects for a class is determined by counting how often a class is listed in the remaining commits. For example, if class A is mentioned in five commits, five defects are assigned to class A. Figure 10 exemplarily shows two commits which contain the word "bug". For the left commit in Fig. 10, the corresponding classes are "CharStreamsTest" and "CharStreams." Hence, both classes are associated with one defect (if these classes are not mentioned in any other commit in the defined range of releases). A quality assessment is then conducted for release 10.0, and the resulting XML file is used for the tool support in order to calculate the risk coefficient for each class. Finally, each class of the software product Google Guava has a computed risk coefficient.

The tool support was configured as follows. The values of the weights, which were presented in Section 4, were used. All findings provided by the PMD and Javadoc analyses were used. Further, all metrics from the complexity factor were included in the analyses.

The FindBugs factor was determined by using the five findings which had the highest number of findings plus findings which were suggested by the reference set, independently of how many findings they had. As already mentioned before, also the weights for the FindBugs factor were not changed. All rules which were violated and provided by the reference set got the weight 1 and the others 0.5.

As stated, the impact factor of the risk coefficient must be typically determined manually (Felderer et al. 2012) and depends strongly on the perspective from which the consequences are determined (Redmill 2004). Thus, the determination of the impact factor is usually done by managers, strategists, customers or users (Redmill 2004). As a result, the impact factor is assumed to a constant value (5) for all classes in this study. We had no sufficient knowledge to determine the impact factor for the different classes and components of each software product. Hence, the risk coefficient in this study is the result of the probability factor multiplied with a constant value. This means the risk coefficient is just a multiple of the probability factor and, therefore, describes the probability whether a class is defect-prone or not. As a , classes with high risk coefficients should have the most defects in it.

5.4 Analysis procedure

The analysis was mainly conducted with quantitative methods. As stated in the previous section, classes with a high risk coefficient should have the most defects in it because the risk coefficient in this study describes only the probability factor as the impact factor is set to a constant value. To answer research question RQ1, the relationship between the risk coefficient and the associated number of defects for each class is analyzed.

In the literature, several methods for analyzing relationships (e.g., logistic regression (Basili et al. 1995) or Spearman rank correlation (Singh et al. 2011)) were proposed. Due to the fact that the tool ranks the classes according to their risk coefficient, an appropriate way of answering research question RQ1 is Spearman's rank correlation coefficient (Spearman 1904). In detail, the range of the Spearman's correlation coefficient (ρ) is between −1 and 1, whereas 1 means the two variables under study are perfectly concordant (positive correlated), −1 means they are perfectly discordand (negatively correlated), and 0 means that there is no relation between the two variables under study (Grzegorzewski and Ziembinska 2011). The values between −1 and 0 as well as between 0 and 1 provide a relative indication of the degree of the correlation between the two variables under study (Grzegorzewski and Ziembinska 2011). In order to determine the significance of the correlation coefficient, the p value must be calculated (McDonald 2014). The p value represents the probability that the results occurred by chance (Spearman 1904; McDonald 2014). We assumed a significance level of 0.05, meaning all results which had a p value smaller than 0.05 were assumed to be significant in the case study.

To address research question RQ2 which deals with comparing a test strategy based on the risk coefficient (risk-based testing strategy) with a test strategy based on the lines of code (lines of code-based testing strategy), the same range of software releases and the same set of determined defects as for research question RQ1 were used.

5.5 Validity procedure

As suggested by Runeson and Höst (2009), threats to validity according to construct validity, reliability as well as conclusion and external validity were analyzed. In order to address common threats to validity, countermeasures were taken. The threats to validity are discussed in Section 6.3.

6 Results and discussion

In this section, we answer the two research questions based on the studied cases. For each research question, we present the results as well as discuss the findings. Finally, threats to validity are discussed. The files which include all data and computations are available at https://git.uibk.ac.at/c703409/sqm-rbt.

6.1 Is there a relationship between the risk coefficient and the number of defects of a class? (RQ1)

To answer the first research question, we analyzed the risk coefficients (impact factor is assumed to be constant) and the number of defects (#defects) for each class of the analyzed software releases. For calculating the Spearman correlation coefficient, first all values of the risk coefficient and all values of the associated number of defects were ranked independently with the same ranking scheme (either from smallest to largest or from largest to smallest) (Grzegorzewski and Ziembinska 2011), for each class. In the case that some classes were ranked equal, each class got an average rank (Sharma 2005).

All five correlation coefficients indicate a low (Taylor 1990) positive correlation and are significant (p value below 0.05). In conclusion, the results show that a positive relationship between the ranking of the classes based on the risk coefficient and the ranking of the classes based on the number of defects exists. Meaning, the higher the rank of a class (based on the risk coefficient), the higher is the associated number of defects for that class.

In addition, we calculated the correlation between the ranking of the classes based on the lines of code (LoC) and the ranking of the classes based on the number of defects (#defects). As expected (according to the literature on software complexity (Section 4.2.1)), the results indicate a low positive and significant correlation between the ranking of the classes based on the lines of code and the ranking of the classes based on the number of defects (correlation coefficients between 0.11 and 0.25).

Table 14 shows the calculated Spearman correlation coefficients (ρ), the associated p values, and the number of observations (n) for both testing strategies (risk-based and line of code-based).

According to these results, both testing strategies seem to be consistent, showing no significant differences referring to the correlation with the number of defects. Therefore, the next subsection compares the performance of both testing strategies in more detail.

Table 14 Spearman's correlation coefficient

Software product	Versions	*risk coeff.* & #defects		*LoC* & #defects		n
		ρ	p value	ρ	p value	
Apache Commons IO	1.4–2.1	0.45	0.00008	0.24	0.041	72
Apache PDFBox	1.0.0–1.8.9	0.18	0.0001	0.11	0.021	433
Google Guava	10.0–18.0	0.21	0.0022	0.24	0.00001	317
JUnit	4.6–4.12	0.16	0.013	0.18	0.0055	247
Mockito	1.0–1.10.19	0.23	0.028	0.25	0.014	95

6.2 How is the performance of a risk-based testing strategy compared with a line of code-based testing strategy? (RQ2)

To answer the second research question, we used the selected software releases which were analyzed by the tool and applied both the risk-based testing strategy and the line of code-based testing strategy. By applying a risk-based testing strategy, testing is started with those classes with the highest risk coefficients (impact factor is assumed to be constant). Hence, all classes are tested, based on their calculated risk coefficients in descending order. In contrast, a line of code-based testing strategy starts testing with those classes with the most lines of code. For both testing strategies, we analyzed how many defects each tested class contained. The tested classes as well as the associated number of defects for each class were further cumulated. To illustrate the results, we used diagrams which show the cumulated number of tested classes on the x-axis and the cumulated associated number of defects on the y-axis. As a result, the diagrams show the distribution of the defects to the cumulative share of tested classes. In the following, we present the results for each of the five analyzed software products. The dashed black line in each diagram represents a testing strategy based on the lines of code, and the solid blue line as testing strategy based on the risk coefficient.

Figure 11 illustrates the two testing strategies for the software product JUnit (release 4.6 with 247 classes). Although a testing strategy based on the lines of code (start testing of classes with the most lines of code) outperforms a testing strategy based on the risk coefficient (start testing of classes with the highest risk coefficient), nearly over the entire testing process, applying a risk-based testing strategy results in finding all defects (6) earlier. Eighty-three percent of all defects are found by testing 13 % of the classes when a risk-based testing strategy is applied, whereas the same amount of defects is found by testing only 6 % of the classes when using a test strategy based on the lines of code.

Also in the case of the software product Mockito (version 1.0 with 95 classes), a test strategy based on the lines of code outperforms at the beginning a risk-based testing strategy. Testing 17 % of all classes results in finding 63 % of the defects when applying a risk-based test strategy and finding 75 % of the defects when applying a lines of code-based testing strategy. For finding all defects (8), a risk-based testing strategy requires to test 44 % of all classes whereas 50 % of all classes are needed to test when applying a line of code-based testing strategy. The diagram with the two testing strategies is shown in Fig. 12.

A comparison of the two testing strategies for the software product Google Guava (version 1.0 with 317 classes) is shown in Fig. 13. The diagram shows a similar trend, as the previous two diagrams, according to the two testing strategies. At the beginning, a line of code-based testing strategy outperforms a risk-based testing strategy. For finding 88 % of all defects, 66 %

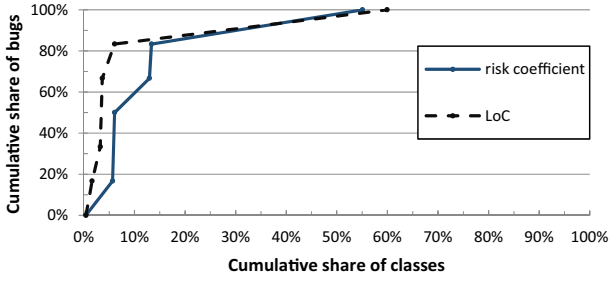


Fig. 11 Risk coefficient vs. LoC – JUnit (r4.6-r4.12)

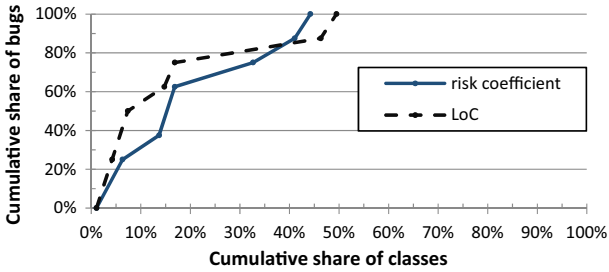


Fig. 12 Risk coefficient vs. LoC – Mockito (v1.0-v1.10.19)

of all classes must be tested when applying a risk-based testing strategy whereas only 52 % must be tested when applying a line of code-based testing strategy. However, for finding all defects (48), the risk-based testing strategy is more efficient than the strategy based on the lines of code.

The analysis of the last two software products, Apache Commons IO (version 1.5 with 72 classes) and Apache PDFBox (version 1.0.0 with 433 classes), revealed a different picture as the previous three diagrams. The comparison of the two testing strategies shows for both software products that a testing strategy based on the risk coefficient outperforms a testing strategy based on the lines of code over the entire testing process. Finding all defects (6) requires testing 13 % of the Apache Commons IO classes for a risk-based testing strategy whereas testing 47 % of the classes is needed when applying a lines of code-based testing strategy. Testing 66 % of Apache PDFBox's classes results in finding all defects (18), whereas 73 % are needed when starting testing with those classes which have the most lines of code. Figures 14 and 15 show the diagrams for the two software products Apache Commons IO and Apache PDFBox.

The results show that, although a testing strategy based on the lines of code outperformed a risk-based testing strategy in three out of five software products nearly over the entire testing process, applying a risk-based testing strategy resulted in finding all defects of the five software products earlier than applying a lines of code-based testing strategy. Table 15 shows the cumulated percentage of classes needed to test for finding all defects for both testing strategies. For finding all defects, on average 51.6 % of the classes must be tested by applying a risk-based testing strategy. In contrast, 61.8 % (on average) of the classes must be tested when applying a line of code-based testing strategy. For finding on average 80 % of all defects, a risk-based testing strategy requires on average test coverage of 30 % (classes).

To summarize, a risk-based testing strategy clearly outperforms a testing strategy based on the lines of code according to find all defects earlier.

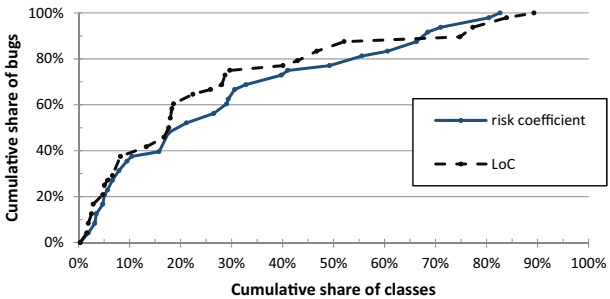


Fig. 13 Risk coefficient vs. LoC - Google Guava (v10.0-v18.0)

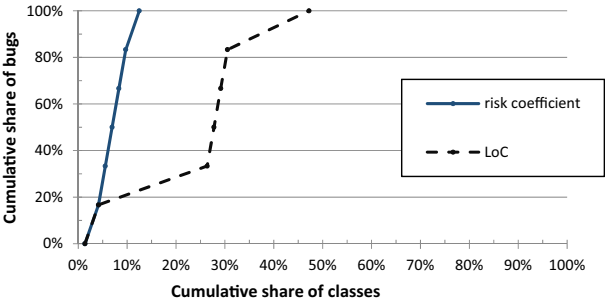


Fig. 14 Risk coefficient vs. LoC - Apache Commons IO (1.4–2.1)

6.3 Threats to validity

In this section, we discuss threats to validity of our results and the applied countermeasures. Referring to Runeson and Höst (2009), we discuss threats to the construct validity, reliability, conclusion validity, and external validity of our case study along with countermeasures taken to reduce the threats.

6.4 Construct validity

Construct validity reflects to what extent the phenomenon under study really represents what the researchers have in mind and what is investigated according to the research questions. In order to avoid threats to construct validity, we first defined and discussed all relevant terms and concepts in Section 2. We explicitly defined risk in the context of risk-based testing as software defects or defects to ensure a common understanding about this term in the case and research context. The developed integration approach is further grounded on the defined terms and concepts. Also, the research questions were then formulated based on these defined terms and concepts. Moreover, the development of the integration approach was mainly done in a methodological way. The proposed factors, metrics, and values of the weights were determined based on the existing body of literature and open source analysis. Nevertheless, one threat to construct validity is the used weight values of the metrics for the complexity factor (1, 0.8, and 0.5). Thus, construct validity should be improved by conducting a sensitivity analysis of the used values of the weights. This also applies for the used weight values of the metrics for the PMD, FindBugs, and Javadoc factors as well as their initial weighting for determining the probability factor.

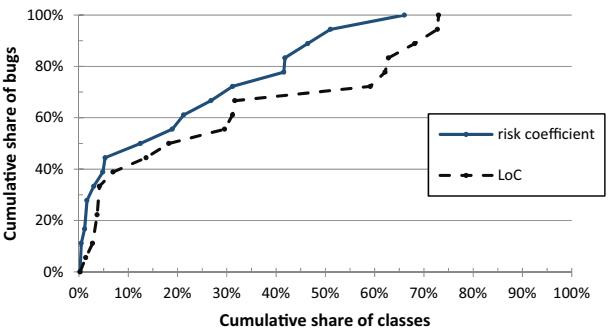


Fig. 15 Risk coefficient vs. LoC - Apache PDFBox (v1.0.0-v1.8.9)

Table 15 Classes needed to test for finding all defects

Software product	Versions	LoC strategy (%)	RBT strategy (%)
Apache Commons IO	1.4–2.1	47	10
Apache PDFBox	1.0.0–1.8.9	73	66
Google Guava	10.0–18.0	89	83
JUnit	4.6–4.12	60	55
Mockito	1.0–1.10.19	50	44
Total average		*63.8*	*51.6*

In addition, GitHub provides a strict guideline how issues should be fixed (GitHub, Inc. 2008c). However, there is no proof that every user uses this procedure. Further, not every issue indicates a defect; there are also other tasks which are addressed by opening and closing issues.

Therefore, all classes which were assumed to be defect-prone were analyzed manually to reduce the amount of false positives. Furthermore, there is no proof that all defects in the software products were closed with a corresponding commit.

Moreover, we only used the word bug to determine the commits which were assumed to deal with defects. Therefore, commits which deal with defects but do not include the word bug in the commit header or message (i.e., commits which use the words "defect," "fault," "failure," or "issue" but not the word bug) were not considered in our analysis. If the commit header or message would also be checked if it contains other words (i.e., "defect," "fault," "failure," or "issue"), the resulting number of commits which had to be analyzed would be too large for a manual analysis. So a manual investigation (as we did in our analysis), if a commit is associated with a real defect, is not feasible when more words are used. As mentioned above, not every commit indicates a defect (there are also other tasks which are addressed by commits). Therefore, the number of false positives would be too large when more words are considered. As a result, we decided to only use the word bug and to check each commit manually in order to focus on real defects. Hence, this is an essential threat to construct validity which must be considered in interpreting the results and should be improved by further work.

In addition, GitHub does not provide any information about the criticality/severity of the committed defects. Hence, an important threat to construct validity is that the defects considered in the study might not have the same level of criticality/severity. Attention in interpreting the results must be paid based on the mentioned facts. Moreover, it is possible that defects in the software products were not fixed before the analysis was conducted.

Lastly, a major threat to construct validity is the fact that we had no suitable knowledge to determine the impact factor. Hence, we assumed the impact factor to a constant value. As a result, we actually use the probability factor (as the risk coefficient is just a multiple) in our case study since the impact factor is constant.

6.5 Reliability

Reliability focuses on whether the data are collected and the analysis is conducted in a way that it can be repeated by other researchers with the same results. This is a threat in any study using qualitative and quantitative data. In order to ensure reliability, the data collection, data processing, and data analyses procedures were well documented. We explicitly documented which software products and releases were

used for the case study and exactly described the procedure for mining the commits (i.e., used words and manual investigation).

6.6 Conclusion validity

Conclusion validity focuses on whether one can be sure that the used treatment of an experiment really is related to the outcome observed. Hence, conclusion validity is of concern when there is a statistically significant effect on the outcome. To address threats according to conclusion validity, we calculated the p value and set the significance level to 5 % in order to minimize the probability that the results occurred by chance.

6.7 External validity

External validity is concerned with to what extent it is possible to generalize the findings and to what extent the findings are of interest to other people outside the investigated cases. Due to the usage of a quality model based on the programming language Java and the analysis of only Java open source software products, one should be careful by generalizing the results on other programming languages like C# or commercial software products. Further, the used quality model provided by the QuaMoCo Tool Chain is a hierarchical quality model. Therefore, the results cannot be generalized without concerns to all groups of quality models. However, the usage of quality assessments based on a hierarchical quality model in risk-based testing is most beneficial because hierarchical quality models are the most predominant group of quality models (Wagner et al. 2015). Moreover, the integration approach is limited to the class testing level. A generalization on other testing levels (i.e., system testing level) seems, based on the results of this case study, promising but must be further investigated.

7 Conclusion and future work

In this article, we explored the integration of quality models and risk-based testing. Therefore, we first presented two generic approaches showing how quality assessments based on quality models can be integrated into risk-based testing. We further illustrated a concrete integration of quality assessments and risk-based testing on the basis of the open quality model QuaMoCo. In addition, we implemented the integration approach as a tool which can be used by practitioners in the risk assessment phase of a risk-based testing process.

A case study of the developed integration approach based on five open source products showed that a risk-based testing strategy outperforms a line of code-based testing strategy according to the number of classes which must be tested in order to find all defects. On average, all defects of the five analyzed software products were found by testing 51.6 % of all classes when a risk-based testing strategy was applied. In contrast, 63.8 % of the classes had to be tested, on average, when a testing strategy based on the lines of code was applied. In addition, a significant positive relationship between the risk coefficient (impact factor assumed to be constant) and the associated number of defects of a class was found. Hence, the case study presented in this article showed a quite sufficient and promising result which constitutes the motivation of the following future work.

First, we intend to perform additional case studies as well as comprehensive field studies, where the developed approach and its tool implementation are applied in an industrial context by

testers. Second, we plan to extend the approach to other programming languages and other types of components besides classes. Third, we want to compare our risk-based testing strategy with other testing strategies (e.g., complexity-based testing strategy). Finally, we plan to improve tool support, for instance with functionality to automatically generate stubs for the tested components.

Acknowledgments Open access funding provided by University of Innsbruck and Medical University of Innsbruck. This work has been supported by the project QE LaB – Living Models for Open Systems (www.qe-lab.at) funded by the Austrian Federal Ministry of Economics (Bundesministerium für Wirtschaft und Arbeit). We thank Stefan Wagner for providing us with infos regarding QuaMoCo.

References

Al-Qutaish, R. E. (2010). Quality models in software engineering literature: an analytical and comparative study. *Journal of American Science, 6*(3), 166–175.

Basili, V. R., Briand, L., & Melo, W. L. (1995). *Technical report (CS-TR-3443, UMIACS-TR-95-40): a validation of object-oriented design metrics as quality indicators*. College Park: University of Maryland, Department of Computer Science.

Bissyande, T. F., Lo, D., Jiang, L., Reveillere, L., Klein, J., & and Le Traon, Y. (2013). *Got issues? Who cares about it? A large scale investigation of issue trackers from GitHub. 24th international symposium on software reliability engineering (ISSRE).* Pasadena: IEEE. Retrieved December 12, 2015, from http://ink.library.smu.edu.sg/sis_research/2087

Boehm, B. W., Brown, J. R., Kaspar, H., Lipow, M., McLeod, G. J., & Merrit, M. J. (1978). *Characteristics of software quality.* Amsterdam: North Holland Publishing.

Catal, C., Diri, & Banu (2007). Software fault prediction with object-oriented metrics based Artifical immune recognition system. In J. Münch & P. Abrahamsson (Eds.), *Product-focused software process improvement, proceedings 8th international conference, PROFES 2007, LNCS 4589* (pp. 300–314). Berlin: Springer.

Cavano, J. P., & McCall, J. A. (1978). A framework for the measurement of software quality. *ACM Sigmetrics Performance Evaluation Review, 7*(3–4), 133–139.

Chacon, S., & Straub, B. (2014). *Pro Git: everything you need to know about Git (Ebook). (2.).* New York City: Apress. Retrieved June 16, 2015, from https://progit2.s3.amazonaws.com/en/2015-05-31-24e8b/progit-en.519.pdf

Checkstyle. (2001). *checkstyle*. Retrieved May 16, 2015, from checkstyle: http://checkstyle.sourceforge.net/

Deissenböck, F., Heinemann, L., Herrmannsdörfer, M., Lochmann, K., & Wagner, S. (2011). *The Quamoco tool chain for quality modeling and assessment. Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, May 21–28* (pp. 1007–1009). Waikiki: ACM.

Deissenböck, F., Juergens, E., Lochmann, K., & Wagner, S. (2009). *Software quality models: purposes, usage scenarios and requirements. ICSE Workshop on Software Quality, WOSQ '09* (pp. 9–14). Vancouver: IEEE Computer Society.

Deissenböck, F., Wagner, S., Pizka, M., Teuchert, S., & Girard, J.-F. (2007). *An activity-based quality model for maintainability. International Conference on Software Maintenance, ICSM 2007* (pp. 184–193). Paris, France: IEEE.

Dixon, M. (2008). *An objective measure of code quality.* Technical report.

Erdogan, G., Li, Y., Runde, R. K., Seehusen, F., & Stolen, K. (2014). Approaches for the combined use of risk analysis and testing: a systematic literature review. *International Journal on Software Tools for Technology Transfer, 16*(5), 627–642.

Felderer, M., & Ramler, R. (2014a). A multiple case study on risk-based testing in industry. *International Journal on Software Tools for Technology Transfer, 16*(5), 609–625.

Felderer, M., & Ramler, R. (2014b). Integrating risk-based testing in industrial test processes. *Software Quality Journal, 22*(3), 543–575.

Felderer, M., & Ramler, R. (2016). Risk orientation in software testing processes of small and medium enterprises: an exploratory and comparative study. *Software Quality Journal*, online first.

Felderer, M., & Schieferdecker, I. (2014). A taxonomy of risk-based testing. *International Journal on Software Tools for Technology Transfer, 16*(5), 559–568.

Felderer, M., Haisjackl, C., Breu, R., & Motz, J. (2012). Integrating manual and automatic risk assessment for risk-based testing. In S. Biffl, D. Winkler, & J. Bergsmann (Eds.), *Software quality. Process automation in software development. SWQD 2012, 17–19 January, Vienna, Austria, LNBIP 94* (pp. 159–180). Berlin: Springer.

Felderer, M., Haisjackl, C., Pekar, V., & Breu, R. (2014a). A risk assessment framework for software testing. In T. Margaria & B. Steffen (Eds.), *Leveraging applications of formal methods, verification and validation: specialized techniques and applications - ISoLA 2014 part II, LNCS 8803* (pp. 292–308). Berlin: Springer.

Felderer, M., Haisjackl, C., Pekar, V., & Breu, R. (2015). An exploratory study on risk estimation in risk-based testing approaches. In D. Winkler, S. Biffl, & J. Bergsmann (Eds.), *Software quality. Software and systems quality in distributed and mobile environments. 7th international conference, SWQD 2015, Vienna, Austria, January 20–23, 2015, Proceedings, LNBIP 200* (pp. 32–43).

Felderer, M., Wendland, M.-F., & Schieferdecker, I. (2014b). Risk-based testing (track introduction). In T. Margaria & B. Steffen (Eds.), *Leveraging applications of formal methods, verification and validation: specialized techniques and applications - ISoLA 2014 part II, LNCS 8803* (pp. 274–276). Berlin: Springer.

FindBugs. (2003). *FindBugs™ - Find bugs in Java programs*. Retrieved May 16, 2015, from FindBugs™: http://findbugs.sourceforge.net/

Franch, X., & Carvallo, J. P. (2003). Using quality models in software package selection. *IEEE Software, 20*(1), 34–41.

Gerrard, P., & Thompson, N. (2002). *Risk-based E-business testing*. Norwood: Artech House Inc..

Git. (2005). *git –distributed-even-if-your-workflow-isnt*. Retrieved June 16, 2015, from git: https://git-scm.com/

GitHub, Inc. (2008b). *GitHub*. Retrieved June 16, 2015, from GitHub: https://github.com/about

GitHub, Inc. (2008c). *Closing issues via commit messages*. Retrieved June 16, 2015, from GitHub: https://help.github.com/articles/closing-issues-via-commit-messages/

Google Inc. (2005). *Google analytics - analysis tools*. Retrieved April 30, 2015, from Google Analytics: http://www.google.com/intl/en_uk/analytics/features/analysis-tools.html

Grady, R. B. (1992). *Practical software metrics for Project Management and process improvement*. New Jersey: Prentice Hill.

Graham, D., Van Veenendaal, E., Evans, I., & Black, R. (2008). *Foundations of software testing: ISTQB certification*. London: Cengage Learning EMEA.

Grzegorzewski, P., & Ziembinska, P. (2011). Spearman's rank correlation coefficient for vague preferences. In H. Christiansen, G. De Tre, A. Yazici, S. Zadrozny, T. Andreasen, & H. L. Larsen (Eds.), *Flexible query answering systems; 9th international conference, FQAS 2011 - Ghent, Belgium, October 2011, Processings; LNAI 7022* (pp. 342–353). Heidelberg: Springer.

Gyimothy, T., Ferenc, R., & Siket, I. (2005). Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering, 31*(10), 897–910.

Huang, F., & Liu, B. (2013). Study on the correlations between program metrics and defect rate by a controlled experiment. *Journal of Software Engineering, 7*(3), 114–120.

ISO/IEC 25010. (2011). *Systems and software engineering – systems and software quality requirements and evaluation (SQuaRE) – System and software quality models.*

ISO/IEC 9126-1. (2001). *ISO/IEC 9126–1*. Retrieved December 10, 2015, from ISO/IEC 9126–1: Software engineering – Product quality: http://www.iso.org/iso/catalogue_detail.htm?csnumber=22749

ISO/IEC/IEEE. (2013). *ISO/IEC/IEEE 29119 Software testing.* Retrieved July 15, 2015, from The International Software Testing Standard: http://www.softwaretestingstandard.org

ISTQB. (2015). *Standard glossary of terms used in software testing - all terms.* Version 3.0, International software testing qualifications board, ISTQB glossary working group. Retrieved April 26, 2015, from ISTQB: http://www.istqb.org/downloads/finish/20/193.html

Jiang, Y., Cukic, B., Menzies, T., & Bartlow, N. (2008). *Comparing design and code metrics for software quality prediction. Proceddings of the 4th international workshop on Predictor models in software engineering, PROMISE '08, 12–13 May, Leipzig, Germany* (pp. 11–18). New York: ACM.

Jureczko, M. (2011). Significance of different software metrics in defect prediction. *Software Engineering: An international Journal (SEIJ), 1*(1), 86–95.

Kitchenham, B., & Pfleeger, S. L. (1996). Software quality: the elusive target. *IEEE Software, 13*(1), 12–21.

Krusko, A. (2003). *Complexity analysis of real time software—using software complexity metrics to improve the quality of real time software.* Master's Thesis in Computer Science, Stockholm, Royal Institute of Technology, KTH Numerical Analysis and Computer Science.

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering, 2*(4), 308–320.

McCall, J. A., Richards, P. K., & Walters, G. F. (1977). *Factors in software quality.* Us Rome Air Development Center. National Technical Information Service.

McDonald, J. H. (2014). *Handbook of biological statistics* (3. ed.). Baltimore: Sparky House Publishing.

Miguel, J. P., Mauricio, D., & Rodriguez, G. (2014). A review of software quality models for the evaluation of software products. *International Journal of Software Engineering & Applications (IJSEA), 5*(6), 31–53.

Nagappan, N., Ball, T., & Zeller, A. (2006). *Mining metrics to predict component failures. Proceddings of the 28th international conference on Software engineering, ICSE '06, 20–28 May, Shanghai, China* (pp. 452–461). New York: ACM.

Neubauer, J., Windmüller, S., & Steffen, B. (2014). Risk-based testing via active continuous quality control. *International Journal on Software Tools for Technology Transfer, 16*(5), 569–591.

Orsini, L. (2013). *GitHub for beginners: don't get scared, get started*. Retrieved June 16, 2015, from readwrite: http://readwrite.com/2013/09/30/understanding-github-a-journey-for-beginners-part-1

Perry, W. E., & Rice, R. W. (1997). *Surviving the top ten challenges of software testing: a people-oriented approach*. New York: Dorset House.

PMD. (2015). *PMD*. Retrieved May 16, 2015, from PMD: http://pmd.sourceforge.net/

Pressman, R. S. (2010). *Software engineering: a Practitioner's approach* (7. ed.). New York: McGraw-Hill.

Pries, K. H., & Quigley, J. M. (2010). *Testing complex and embedded systems*. Boca Raton: CRC Press.

Radjenovic, D., Hericko, M., Torkar, R., & Zivkovic, A. (2013). Software fault prediction metrics: a systematic literature review. *Information and Software Technology, 55*(8), 1397–1418.

Redmill, F. (2004). Exploring risk-based testing and its implications. *Software Testing, Verification and Reliability, 14*(1), 3–15.

Redmill, F. (2005). Theory and practice of risk-based testing. *Software Testing, Verification and Reliability, 15*(1), 3–20.

Runeson, P., & Höst, M. (2009). Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering, 14*(2), 131–164.

Sharma, A. K. (2005). *Text book of correlations and regression*. New Delhi: Discovery Publishing House.

Singh, P., Chaudhary, K. D., & Verma, S. (2011). An investigation of the relationships between software metrics and defects. *International Journal of Computer Applications, 28*(8), 13–17.

Spearman, C. (1904). The proof and measurement of association between two things. *The American Journal of Psychology, 15*(1), 72–101.

Taylor, R. (1990). Interpretation of the correlation coefficient: a basic review. *Journal of Diagnostic Medical Sonography, 6*(1), 35–39.

Van Veenendaal, E. (2009). *Practical risk-based testing - product RISk MAnagement: the PRISMA ® method.* Improve Quality Services BV.

Van Veenendaal, E. (2012). *The PRISMA approach: practical risk-based testing.* UTN Publishers.

Wagner, S. (2013). *Software product quality control*. Berlin: Springer.

Wagner, S., Lochmann, K., Heinemann, L., Kläs, M., Trendowicz, A., Plösch, R.,. . . Streit, J. (2012b). *The Quamoco product quality modelling and assessment approach. 34th international conference on software engineering (ICSE), 2012*, (pp. 1133–1142). Zürich.

Wagner, S., Lochmann, K., Winter, S., Deissenböck, F., Jürgens, E., Herrmannsdörfer, M.,. . . Kläs, M. (2012c). *The Quamoco quality meta-model.* Technischer Bericht TUM-I128, Technische Universität München, Institut für Informatik.

Wagner, S., Goeb, A., Heinemann, L., Kläs, M., Lampasona, C., Lochmann, K., et al. (2015). Operationalised product quality models and assessment: the Quamoco approach. *Information and Software Technology, 62*, 101–123.

Wagner, S., Lochmann, K., Winter, S., Goeb, A., Kläs, M., & Nunnenmacher, S. (2012a). *Software quality models in practice.* Institut für Informatik, TUM-I129. Technische Universität München. Retrieved December 08, 2015, from https://mediatum.ub.tum.de/doc/1110601/1110601.pdf

Windmüller, S., Neubauer, J., Steffen, B., Howar, F., & Bauer, O. (2013). *Active continuous quality control. Proceedings of the 16th International ACM Sigsoft symposium on Component-based software engineering (CBSE '13), June 17–21, 2013, Vancouver, BC, Canada* (pp. 111–120). New York: ACM.

Zeiss, B., Vega, D., Schieferdecker, I., Neukirchen, H., & Grabowski, J. (2007). Applying the ISO 9126 quality model to test specifications - exemplified for TTCN-3 test specifications. *Software Engineering, 15*(6), 231–242.

Zhang, H. (2009). *An investigation of the relationships between lines of code and defects. International conference on software maintenance 2009, ICSM 2009* (pp. 274–283). IEEE.

Zhang, Q., Wu, J., & Zhu, H. (2006). *Tool support to model-based quality analysis of software architecture. Proceedings of the 30th annual international computer software and applications conference (COMPSAC'06)* (pp. 121–128). IEEE Computer Society.

Zimmermann, T., Nagappan, N., & Zeller, A. (2008). Predicting bugs from history. In T. Mens & S. Demeyer (Eds.), *Software evolution* (pp. 69–88). Berlin: Springer.

Zimmermann, T., Premraj, R., & Zeller, A. (2007). *Predicting defects for eclipse. Third International Workshop on Predictor Models in Software Engineering (PROMISE'07), 20–26 May 2007* (pp. 9–16). Minneaplois: IEEE Computer Society.

Harald Foidl is a Ph.D. student in computer science at the University of Innsbruck. He received a MSc in information systems in 2015 from the University of Innsbruck. Beside his studies, Harald worked as a programmer and test engineer for the company exceet electronics GesmbH. His research interests include software engineering and testing of safety critical systems, data modeling as well as data mining. In addition to his research activities, Harald is currently head of the Surface-Mount Technology Department of the company exceet electronics GesmbH.

Michael Felderer is a senior researcher and project manager at the Institute of Computer Science at the University of Innsbruck, Austria. He holds a Ph.D. and a habilitation degree in computer science. His research interests include software testing and software quality in general, requirements engineering, empirical software engineering, software processes, security engineering, and improving industry-academia collaboration. He works in close collaboration with industry and transfers his research results into practice as a consultant and speaker on industrial conferences.